A HARDWARE/SOFTWARE CO-DESIGN APPROACH FOR FACE RECOGNITION BY ARTIFICIAL NEURAL NETWORKS

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

XIAOGUANG LI

In partial fulfilment of requirements

for the degree of

Masters of Science

August, 2004

ABSTRACT

A HARDWARE/SOFTWARE CO-DESIGN APPROACH FOR FACE RECOGNITION BY ARTIFICIAL NEURAL NETWORKS

Xiaoguang Li
University of Guelph, 2004

Advisor:
Dr. Shawki M. Areibi

Artificial Neural Networks (ANNs), and the multi-layer perceptrons trained using an error *backpropagation* algorithm (MLP-BP) in particular, have proven to be an effective method today in many applications. However, this technique has suffered from slow training and lack of clear methodology to determine the network topology before training starts. The speedup to this algorithm is desired so that reasonable experimentation with various network topologies and on-line working are possible. Although Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) can achieve speedup over a general processor, the flexibility is a tradeoff with speed. To balance them, an embedded computing system consisting both a processor with dedicated hardware on an FPGA chip is proposed. Results obtained show that this system achieves 1.69 speedup (Amdahl's Law) over the system which consists of only a processor on an FPGA chip. At the same time, the flexibility is preserved to some extent.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation and thanks to my adviser, Dr. Shawki M. Areibi for his great help and guidance, and also the inspiration he provided me at difficult times. Without his support, criticism, and invaluable help, this work would never have been possible. I would also like to thank Dr. Medhat Moussa for his great guidance in investigating the effect of different arithmetic formats in implementing Artificial Neural Networks on FPGAs. This work provided a very good beginning to my research.

Special thanks to Dr. Bob Dony for his invaluable inspiration in the area of Digital Signal Processing and great encouragement in the accomplishment of this thesis.

Great thanks to my wife Xiaoyan Xu and my parents for their encouragement and support at those difficult times.

To

my family

whose love and encouragement helped accomplish this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An Artificial Neural Network (ANN) is an information-processing paradigm that is inspired by the biological nervous system, i.e. the human brain. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected parallel processing elements (neurons) working together to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition [Song97] or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. The most common architecture in ANNs consists of multi-layer perceptrons trained using an error *backpropagation* algorithm (MLP-BP) [Rume86]. One of the main problems in training such a network is the lack of a clear methodology to determine the network topologies before training starts. Experimenting with various topologies is difficult due to the long time required for each training session, especially with large networks. The network topology is an important factor in the network's ability to generalize after the training is completed. A larger than needed network may over-fit the training data and results in poor generalization on the testing data, while a smaller than needed network may not have the computational capacity to approximate the target function. Furthermore, in applications where on-line training is required, the training time is often a critical parameter. For these reasons, it is desirable to speed up the latter. This allows for reasonable experimentation with various network topologies and the ability to use them in on-line applications.

1.1 Motivation and Objectives

ANNs have broad applicability to real world problems. In fact, they have already been successfully applied in many industries. Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs, including sales forecasting, industrial process control, customer research, data validation, risk management, target marketing, etc. To be more specific, ANNs are used in the following specific paradigms: texture analysis [CBus93]; three-dimensional object recognition [Rosa94]; hand-written word recognition [Blum98, BV98]; loan underwriting [EC88]; medical diagnoses [DGB88]; and face recognition [Toh02, Lin97, Lawr97].

1.1.1 Motivation

Face recognition is employed in large-scale citizen identification applications, surveillance applications, law enforcement applications such as booking stations, and kiosks. ANNs have proven to be an effective way to solve the face recognition problem, but because of the long time training process, this approach is not suitable for real-time applications which are desirable in most face recognition applications. In order to accelerate the training process, hardware like ASICs or FPGAs is used. On the other hand, we still need the flexibility to some extent. Using a Hardware/Software Co-design methodology to design a system containing both dedicated hardware and software (i.e. a processor) is preferred. Today, the capacity of FPGAs has reached the point where both software and dedicated hardware can be implemented simultaneously. This is an excellent platform for implementing ANNs. In this way, the most time-consuming part of the MLP-BP algorithm could be implemented by programmable logic of a FPGA to achieve acceleration, while the rest could be implemented by software (i.e., a processor implemented inside the same FPGA) to preserve the flexibility.

1.1.2 Objectives

The objective of this thesis is to investigate flexibility vs. performance tradeoff in implementing MLP-BP onto a Reconfigurable Computing (RC) platform (i.e. FPGA) for solving the face recognition problem. To reach the overall goal, the objectives were identified as follows:

- To evaluate floating-point vs. fixed-point operations from RC point of view. Identifying the type of operations used is very crucial in RC implementation of ANNs.

- To evaluate a pure *MicroBlaze* [Micr] processor system on a Virtex-II FPGA developing board with the functionality of solving face recognition problem by an ANSI C program on a *MicroBlaze* processor for the *backpropagation* algorithm.

- To evaluate an Embedded computing system on the same Virtex-II FPGA developing board with the functionality of solving face recognition problem by ANSI C program on *MicroBlaze* for part of the *backpropagation* algorithm and dedicated hardware for the rest of the *backpropagation* algorithm.

1.2 Overview of the research work

The overall research path presented in this thesis is illustrated in Figure 1.1.

To solve the face recognition problem by an ANN, first a software implementation was evaluated on a PC workstation. From this evaluation, several drawbacks were identified including the long processing time required in the training phase, etc. In order to overcome these problems, the FPGA design style was chosen as a means to achieve speedup. Prior to this, a basic investigation of different arithmetic formats for implementing ANNs on FPGAs was conducted. Some interesting results were identified in that floating-point operators were feasible for implementing ANNs. Two implementations that tradeoff flexibility and performance were designed. The first implementation used a soft processor core implemented onto the FPGA to achieve the functionality of a MLP-BP ANN in solving the face recognition problem. In the second implementation, a hardware/software approach utilizing a soft core and dedicated hardware modules was used. In this implementation, the dedicated hardware is designed to execute of the most time-consuming part of the

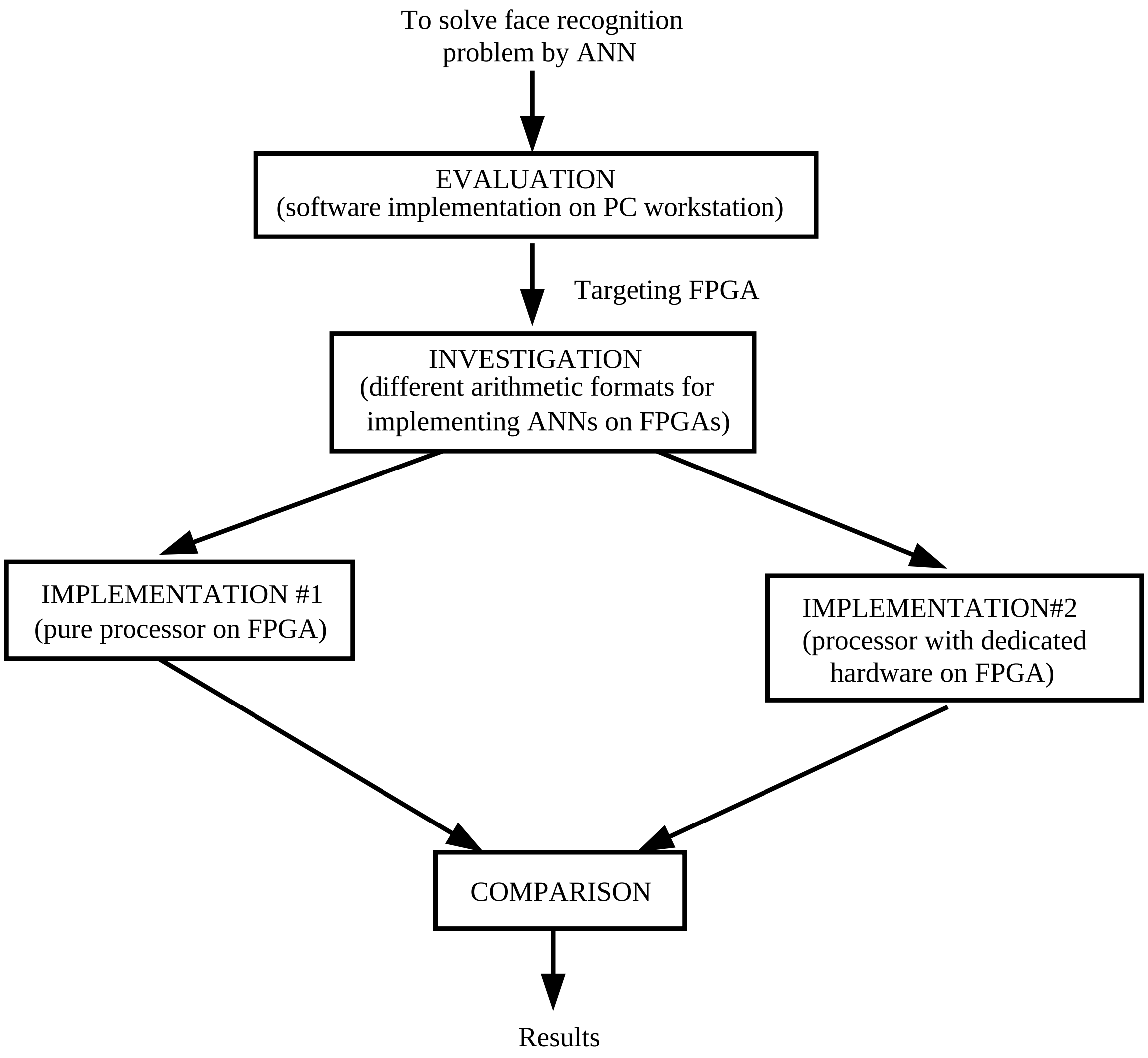
To solve face recognition problem by ANN
EVALUATION (software implementation on PC workstation)
Targeting FPGA
INVESTIGATION (different arithmetic formats for implementing ANNs on FPGAs)
IMPLEMENTATION #1 (pure processor on FPGA)
IMPLEMENTATION#2 (processor with dedicated hardware on FPGA)
COMPARISON
Results

Figure 1.1: Research Path

MLP-BP and the processor is used for the rest of the MLP-BP. The comparison drawn on these two implementations shows that the second implementation has a better performance in terms of speed meanwhile the flexibility is preserved to a certain degree.

1.3 Contributions

The main contributions can be summarized as follows:

- Investigating different arithmetic architectures and formats for implementing ANNs on FPGAs.
- Construction of two embedded computing systems for implementing ANNs in solving the face recognition problem.
- Publication of the above mentioned results in the Canadian Journal of Electrical and Computer Engineering (CJECE) [XL].

  Research performed in this thesis should allow other researchers interested in accelerating learning by ANN to effectively map it onto any RC platform (i.e., FPGAs).

1.4 Thesis outline

Chapter 2 introduces the necessary background used in this thesis including ANNs, FPGAs, Hardware/Software Co-design, and the face recognition problem. **Chapter 3** provides an overview of previous efforts at implementing ANNs on FPGAs and ASICs. **Chapter 4** presents a study on the effect of different arithmetic formats for implementing ANNs on FPGAs. **Chapter 5** presents the proposed approach to implementing an MLP-BP ANN for solving the face recognition problem. The thesis concludes in **Chapter 6** with possible directions for future work.

Chapter 2

Background

In this Chapter, all necessary background related to Artificial Neural Networks (ANNs), Field Programmable Gate Arrays (FPGAs), and Hardware/Software Co-design will be introduced.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are intelligent information processing systems that are inspired by the biological nervous systems (i.e., the human brain). In human brains, a typical neuron shown in Figure 2.1 collects signals from other cells through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activities through long, thin structures known as *axons*, which split into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the *axon* into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.

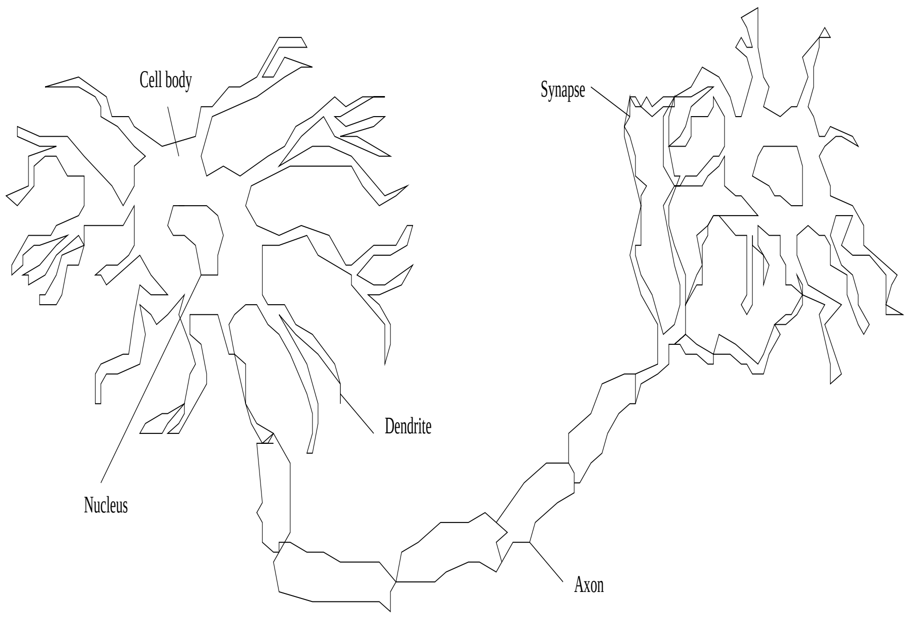

Figure 2.1: A Biological Neuron

2.1.1 From Human Neurons to Artificial Neurons

Inspired by a biological neuron, an artificial neuron is constructed as shown in Figure 2.2. All information collected by *dendrites* is accumulated and then followed by a threshold function, and finally sent through the *axon*.

ANNs are typically composed of interconnected units which serve as the artificial neuron shown in Figure 2.2. The function of the synapse is modeled by a modifiable weight, which is associated with each connection. Each unit converts the pattern of

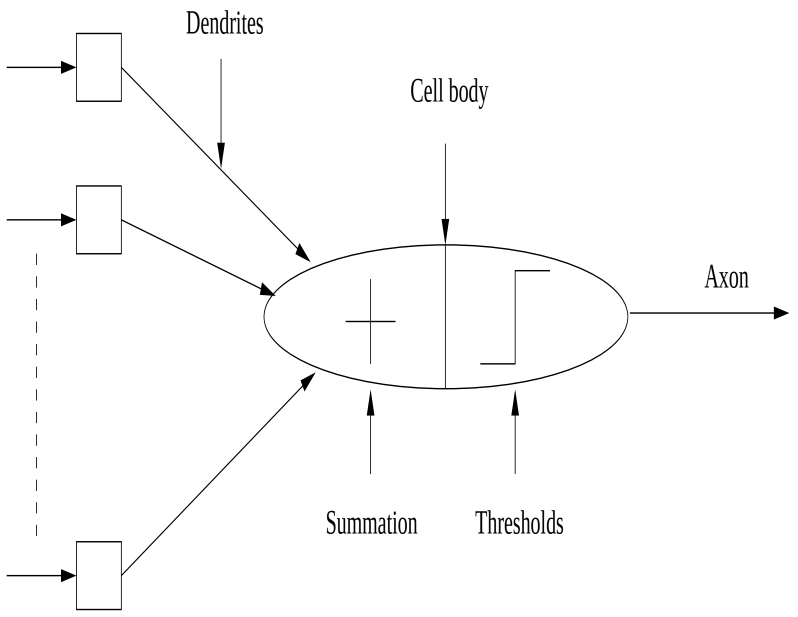

Figure 2.2: An Artificial Neuron Model

incoming activities that it receives into a single outgoing activity that it broadcasts to other units. It performs this conversion in two stages shown in Figure 2.3:

- It multiplies each incoming activity by the weight on the connection and sums all these weighted inputs to get a quantity called the *total input*.
- A unit uses an input-output transfer function that transforms the total input into the outgoing activity.

The behaviour of an ANN depends on both the weights and the input-output function (transfer function) that is specified for the units. Typically, a sigmoid function is used, in which the output varies continuously but not linearly as the input changes.

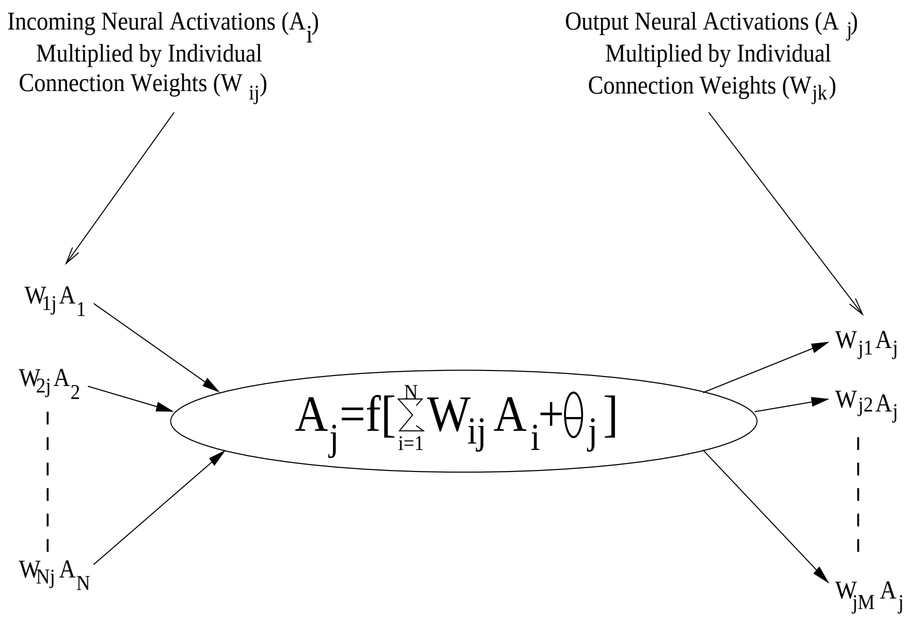
Incoming Neural Activations (A_i) Multiplied by Individual Connection Weights (W_{ij})
Output Neural Activations (A_j) Multiplied by Individual Connection Weights (W_{jk})
$W_{1j} A_1$
$W_{2j} A_2$
$W_{Nj} A_N$
$A_j = f[\sum_{i=1}^{N} W_{ij} A_i + \theta_j]$
$W_{j1} A_j$
$W_{j2} A_j$
$W_{jM} A_j$

Figure 2.3: A Processing Element within a Neuron

Sigmoid hidden and output units usually use a "threshold" term (θ_j) of Figure 2.3 in computing the net input to the unit.

Consider a multi-layer perceptron with any of the usual sigmoid activation functions. Chosen any hidden unit or output unit, if there are N inputs to that unit, which define an N-dimensional space, the given unit draws a hyperplane through that space, producing an "on" output on one side and an "off" output on the other.

The weights determine where this hyperplane lies in the input space. Without a threshold term, this separating hyperplane is constrained to pass through the origin of the space defined by the inputs. For some problems that's OK, but in many problems the hyperplane would be much more useful somewhere else. If you have many units in a layer, they share the same input space and without threshold they would all be constrained to pass through the origin.

To construct a neural network that performs a specific task, we have to determine the connectivity among neurons and identify the appropriate weights of the connections. The connection determines the influence of a neuron over another, whereas the weights specify the strength of the influence.

The most common type of ANN is a multi-layer perceptron structure shown in Figure 2.4. The network consists of one *input* layer, several *hidden* layers and an *output* layer.

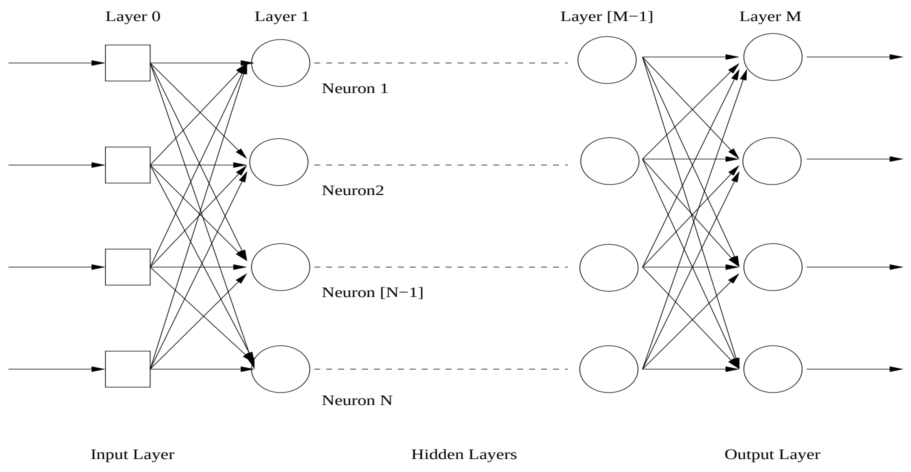

Figure 2.4: The Multi-layer Perceptron Structure ANN

- The activity of the units in the input layer represents the raw information which is fed into the network.

- If the unit is in the hidden layer right after the input layer, the activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input layer and the current hidden layer. If the unit is in other hidden layers, the activity of each hidden unit is determined by the activities of the previous hidden layer and the weights on the connections between the previous hidden and the current hidden units.

- The behaviour of the output units depends on the activity of the previous hidden units and weights between the previous hidden and output units.

This network may be trained to perform a particular task by the following procedure:

- presenting the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of the activities for the output units.
- determining how closely the actual output of the network matches the desired output.
- changing the weight of each connection so that the network produces a better approximation of the desired output.

  To train an ANN to perform some tasks, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. Among the learning algorithms, *backpropagation* algorithm is the most widely used training method which is introduced next.

2.1.2 The *Backpropagation* Algorithm

1. **Encoding:** Figure 2.4 shows a multi-layer perceptron ANN architecture. Each circle of the network is a neuron node or a processing element (PE).

The input stimulus comes to the input nodes of ANN (left side of Figure 2.4) and emerges at the output nodes (right side of Figure 2.4). The input to output mapping is performed by minimizing a *cost function*. The weight connections and threshold adjustments are made according to the error between the computed and desired output PE values. Usually, the *squared error* is used, which is the squared difference between the computed and desired output values for each output PE across all patterns in the training set. Other cost functions which can be employed are the entropic cost function, the linear error, and the Minkowski-r backpropagation (the r^{th} power of the absolute value of the error).

The weight adjustment procedure is derived by computing the change in the cost function with respect to the change in each weight and threshold. This derivation is extended to find the equation for adapting the connections between the input and the hidden layers.

In order to explain *backpropagation* algorithm [Hayk99] in detail, some notations should be introduced. Using a three-layer perceptron ANN as an example:

Notation

- The indices i, h and o refer to different neurons in the network. Neuron i lies in the input layer, neuron h lies in the hidden layer, and neuron o lies in the output layer.
- a_i refers to the value of i^{th} input node.
- b_h refers to the value of h^{th} hidden node.

- c_o refers to the value of o^{th} output node, and c_o^k refers to the desired value of o^{th} output node.
- d_o refers to error at the output of neuron o, and e_h refers to the error at the output of neuron h.
- θ_h and θ_o refer to the value of threshold of neuron h and o.
- E refers to the instantaneous sum of error squares.
- v_{ih} denotes the synaptic weight connecting input layer (i) to hidden layer (h). $\triangle v_{ih}$ is the correction applied to this weight.
- w_{ho} denotes the synaptic weight connecting hidden layer (h) to output layer (o). $\triangle w_{ho}$ is the correction applied to this weight.
- The activation function describing the input-output functional relationship of the nonlinearity is denoted by $f(\cdot)$.
- α is the learning rate. This parameter is used to control the speed of adjustment to neuron parameters (i.e. neuron weights or thresholds).

2. **The BP Training Algorithm:**

   (a) **Error:** The activation value of a hidden unit is given by

$$b_h = f(\sum_i v_{ih} a_i + \theta_h) \tag{2.1}$$

where b_h is the activation value of a hidden layer unit. The activation value for an output unit is

$$c_o = f(\sum_h w_{ho} b_h + \theta_o) = f(\sum_h w_{ho} f(\sum_i v_{ih} a_i + \theta_h) + \theta_o) \tag{2.2}$$

where c_o is the activation value of an output layer unit. The *error* or discrepancy between the calculated and desired value of an output layer unit can be calculated as following:

$$E = \frac{1}{2} \sum_o [c_o^k - c_o]^2 = \frac{1}{2} \sum_o [c_o^k - f(\sum_h w_{ho} f(\sum_i v_{ih} a_i + \theta_h) + \theta_o)]^2 \tag{2.3}$$

This is a *continuous, differentiable* function and, therefore, we can perform a *gradient descent*.

(b) **From Hidden Layer to Output Layer:** First, let us look at the weight changes on the connections from the hidden layer to the output layer over p input-output training pairs.

$$\begin{aligned} \triangle w_{ho} &= -\eta \frac{\delta E}{\delta w_{ho}} && (2.4) \\ &= \eta \sum^p [c_o^k - c_o] f'(\sum_h w_{ho} b_h + \theta_o) b_h && (2.5) \\ &= \eta \sum^p d_o b_h && (2.6) \end{aligned}$$

$$\triangle \theta_o = -\eta \frac{\delta E}{\delta \theta_o} \tag{2.7}$$

$$= \eta \sum^{p} [c_o^k - c_o] f'(\sum_h w_{ho} b_h) \tag{2.8}$$

$$= \eta \sum^{p} d_o \tag{2.9}$$

where

$$d_o = f'(\sum_h w_{ho} b_h + \theta_o)[c_o^k - c_o] \tag{2.10}$$

In summary, the weight change can be written as

$$\triangle w_{ho} = \alpha d_o b_h \tag{2.11}$$

$$\triangle \theta_o = \alpha d_o \tag{2.12}$$

If the threshold function is the sigmoid $f(x) = \frac{1}{1+e^{-x}}$ then the derivative of this function can be expressed in terms of itself as $d_o = c_o(1-c_o)(c_o^k - c_o)$, and so the change in connection weight can be expressed as

$$\triangle w_{ho} = \alpha[c_o(1 - c_o)(c_o^k - c_o)]b_h \tag{2.13}$$

(c) **From Input Layer to Hidden Layer:** Weight changes on the connections from the input layer to the hidden layer are calculated by differentiating E with respect to v_{ih}, using the chain rule:

$$\begin{aligned}
\triangle v_{ih} &= -\eta\frac{\delta E}{\delta v_{ih}} = -\eta\sum^{p}\frac{\delta E}{\delta b_h}\frac{\delta b_h}{\delta v_{ih}} && (2.14)\\
&= \eta\sum^{p}[c_o^k - c_o]f'(\sum_h w_{ho}b_h + \theta_o)w_{ho}f'(\sum_i v_{ih}a_i + \theta_h)a_i && (2.15)\\
&= \eta\sum^{p} d_o w_{ho} f'(\sum_i v_{ih}a_i + \theta_h)a_i && (2.16)\\
&= \eta\sum^{p} e_h a_i && (2.17)
\end{aligned}$$

and

$$\begin{aligned}
\triangle \theta_h &= -\eta\frac{\delta E}{\delta \theta_h} = -\eta\sum^{p}\frac{\delta E}{\delta b_h}\frac{\delta b_h}{\delta \theta_h} && (2.18)\\
&= \eta\sum^{p}[c_o^k - c_o]f'(\sum_h w_{ho}b_h + \theta_o)w_{ho}f'(\sum_i v_{ih}a_i + \theta_h) && (2.19)\\
&= \eta\sum^{p} d_o w_{ho} f'(\sum_i v_{ih}a_i + \theta_h) && (2.20)\\
&= \eta\sum^{p} e_h && (2.21)
\end{aligned}$$

where

$$e_h = f'(\sum_i v_{ih}a_i + \theta_h)\sum_h w_{ho}d_o \qquad (2.22)$$

Thus, the weight change is $\triangle v_{ih} = \beta a_i e_h$ and $\triangle \theta_h = \beta e_h$ once again by using the *Logistic* threshold function:

$$e_h = b_h(1 - b_h) \sum_h w_{ho} d_o \tag{2.23}$$

and

$$\triangle v_{ih} = \beta a_i [b_h(1 - b_h) \sum_h w_{ho} c_o (1 - c_o)(c_o^k - c_o)] \tag{2.24}$$

(d) **In General...**

The general form of a weight change is

$$\triangle w_{pq} = \eta \sum_{patterns} d_{OUTPUT} \times V_{INPUT} \tag{2.25}$$

where d_{OUTPUT} depends on the layer and

- the last layer uses Equation 2.10, and
- the other layers use Equation 2.22.

and V_{INPUT} represents the appropriate input-end activation.

(e) **The Threshold Function**

Now let us examine the threshold function in detail. The general form of the *Logistic* function is the following equation:

$$f_\beta(x) = \frac{1}{1 + e^{-2\beta x}} \tag{2.26}$$

where β is a **steepness** parameter (often $\frac{1}{2}$ or 1). The derivative of this function is

$$f'_\beta = 2\beta f(1-f) \tag{2.27}$$

Now if the **steepness** parameter is $\frac{1}{2}$ then $f'(x) = f(1-f) = c_j(1-c_j)$. The plot of a *Logistic* function is plotted in Figure 2.5 .

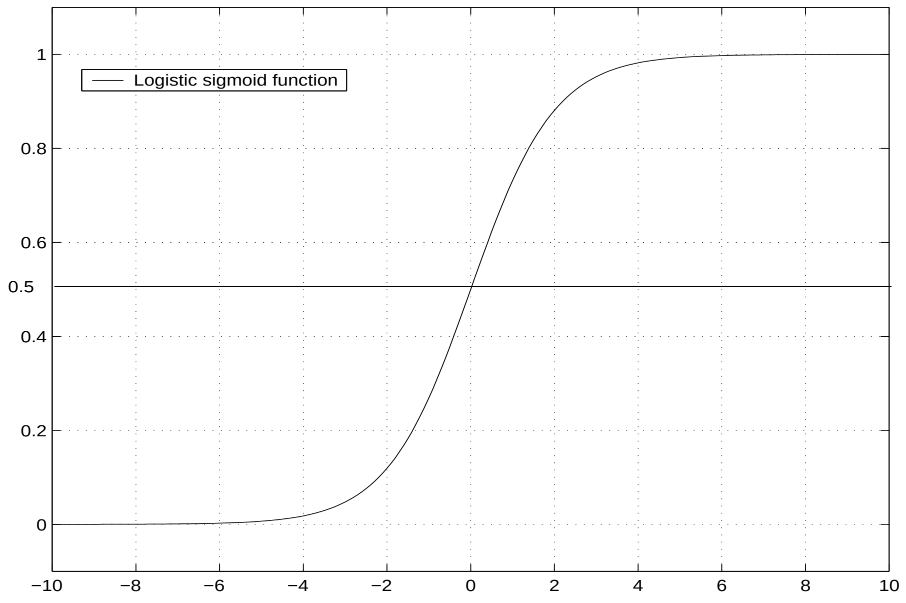

Figure 2.5: The Logistic Sigmoid Function Plot

The general form of the *Hyperbolic Tangent* function is

$$\begin{aligned} f_\beta(x) &= tanh \beta x \qquad (2.28) \\ &= \frac{(1-e^{-\beta x})}{(1+e^{-\beta x})} \qquad (2.29) \end{aligned}$$

The derivative of this function is

$$f'_\beta(x) = \beta(1 - f^2) \tag{2.30}$$

If $\beta = 1$ then $f'(x) = (1 - c_j^2)$ and it is plotted in Figure 2.6.

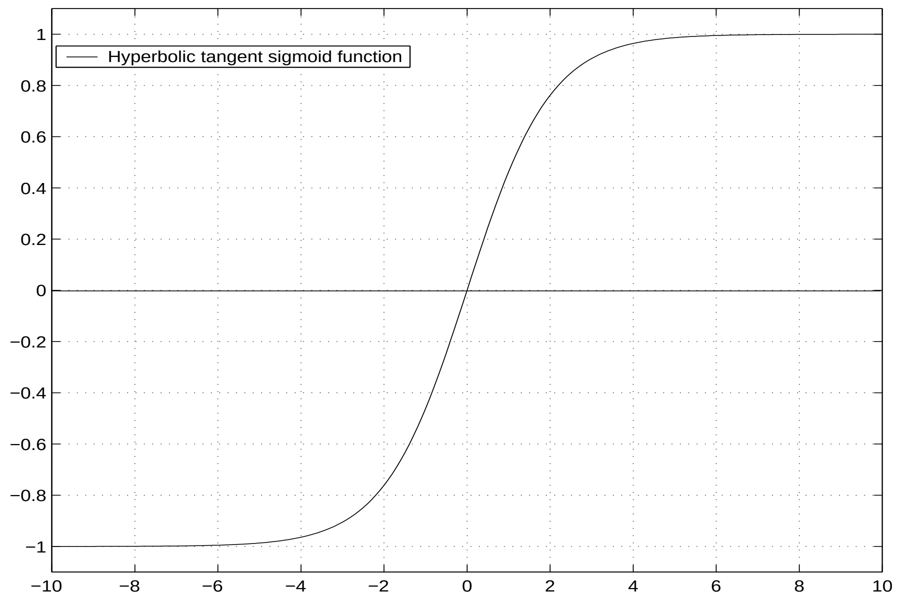

Figure 2.6: The Hyperbolic Tangent Sigmoid Plot

As you can see in Figure 2.5 and Figure 2.6, *Hyperbolic Tangent* function and *Logistic* function are all sigmoid curves, but the output ranges offered are different. The *Hyperbolic Tangent* function offers a larger output (-1,1) range than the *Logistic* function (0,1). In practice, for binary (0/1) targets, the *Logistic* function is a good choice [Jord95], for continuous-valued targets with a bounded range, the *Logistic* and *Hyperbolic Tangent* functions can be used, provided you either scale the outputs to the range of the targets or scale the targets to the range of the output activation function ("scaling" means multiplying by and adding appropriate constants).

3. **Vanilla Version *Backpropagation*** Among the different versions of *backpropagation* algorithm, the *vanilla version backpropagation* is popular and easy to implement. Due to these advantages, it is chosen as the one implemented in this thesis. It minimizes the **squared error cost function** and uses **three-layer** elementary *backpropagation* topology. It is known as the *generalized delta rule.*

   (a) **The Encoding Algorithm**

      i. Assign random values in the range [+1,-1] to all input to hidden connections, v_{ih}, all hidden to output connections, w_{ho}, to each hidden processing element (PE) threshold, θ_h, and to each output PE threshold, θ_o.

      ii. For each pattern pair (A_k, C_k), do the following:

         - Process A_k's values to calculate the new hidden layer PE activations using:

$$b_h = f(\sum_{i=1}^{n} a_i v_{ih} + \theta_h) \tag{2.31}$$

where $f(\cdot)$ is the *Logistic* sigmoid threshold function

$$f(x) = (1 + e^{-x})^{-1} \tag{2.32}$$

- Filter the hidden layer activation to the output layer using

$$c_o = f(\sum_{h=1}^{n} b_h w_{ho} + \theta_o) \tag{2.33}$$

- Compute the error between computed and desired output PE value using

$$d_o = c_o(1 - c_o)(c_o^k - c_o) \tag{2.34}$$

- Calculate the error of each hidden PE relative to each d_o with

$$e_h = b_h(1 - b_h) \sum_{o=1}^{q} w_{ho} d_o \tag{2.35}$$

- Adjust the hidden to output connections by

$$\triangle w_{ho} = \alpha b_h d_o \tag{2.36}$$

  where δw_{ho} is the amount of change made to the connection from the h^{th} hidden layer PE to the o^{th} output layer PE and α is a positive constant controlling the learning rate.

- Adjust the output layer threshold by

$$\triangle \theta_o = \alpha d_o \tag{2.37}$$

- Adjust the input to hidden connections by

$$\triangle v_{ih} = \alpha a_i e_h \tag{2.38}$$

where α is a positive constant controlling the learning rate.

- Adjust the hidden layer thresholds by

$$\triangle \theta_h = \alpha e_h \tag{2.39}$$

- Repeat step 2 until d_o are all either zero or sufficiently low.

iii. **The Recall Mechanism** The recall mechanism consists of two feed-forward operations:

A. create the hidden layer PE values

$$b_h = f(\sum_{i=1}^{n} a_i v_{ih} + \theta_h) \tag{2.40}$$

B. after all hidden layer PE activations have been calculated they are used to create new output layer PE values

$$c_o = f(\sum_{h=1}^{n} b_h w_{ho} + \theta_o) \tag{2.41}$$

iv. **Convergence** *backpropagation* is not guaranteed to find the global error minimum during training, only the local error minimum. But empirical evidence has shown that this is not much of a problem for most practical applications of the algorithm. Simple gradient descent is very slow because we have information only about one point and no clear picture of how the cost surface may curve. The use of small steps takes forever, but big steps may cause divergent oscillations across ”ravines” in the cost surface.

The successful employment of the multi-layer perceptron using *back-propagation* learning entails a consideration of many factors:

- the number of hidden layer PEs,
- the size of the learning rate parameters, and
- the amount of data necessary to create the proper mapping.

A three-layer ANN using *backpropagation* can approximate a wide range of functions to any desired degree of accuracy. If a mapping exists, then it can usually find it.

2.2 Field Programmable Gate Arrays (FPGAs)

FPGAs are chips which are programmed by the customer to perform the desired functionality. The chip may be programmed once (Anti-fuse technology), several times (Flash technology), or dynamically (SRAM technology).

- ***Anti-fuse* FPGAs:** devices are configured by burning a set of fuses. Once the chip is configured, it cannot be altered.
- ***Flash* FPGAs:** devices may be re-programmed several thousand times and are non-volatile, i.e., keep their configuration after power-off.
- ***SRAM* FPGAs:** currently this kind of FPGA is the dominating technology, and they can be re-programmed without limit. In order to load the

configuration into FPGA after power-on, additional circuitry is required. Re-configuration is very fast, and some devices even allow partial re-configuration during operation.

2.2.1 Architecture of FPGAs

Several families of FPGAs are available from different semiconductor companies. These device families differ slightly in their architecture and feature set. However most of them follow a common approach: A regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs) surrounded by a perimeter of programmable Input/Output Blocks (IOBs). These functional elements are interconnected by a powerful hierarchy of versatile routing channels.

The architecture implemented by ***Xilinx Virtex-II FPGAs*** is shown in Figure 2.7, since this FPGA is used in this thesis. In this block diagram, four major elements are organized in a regular array:

- **CLBs** provide the functional elements for the combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.
- **Block SelectRAM** provides large 18 Kbit storage elements of dual-port RAM.
- **Multiplier blocks** are 18-bit $\times$ 18-bit dedicated multipliers.
- **Digital Clock Manager (DCM)** provides a self-calibrating, fully digital

solution for clock distribution delay compensation, clock multiplication and division, and coarse- and fine-grained clock phase shifting.

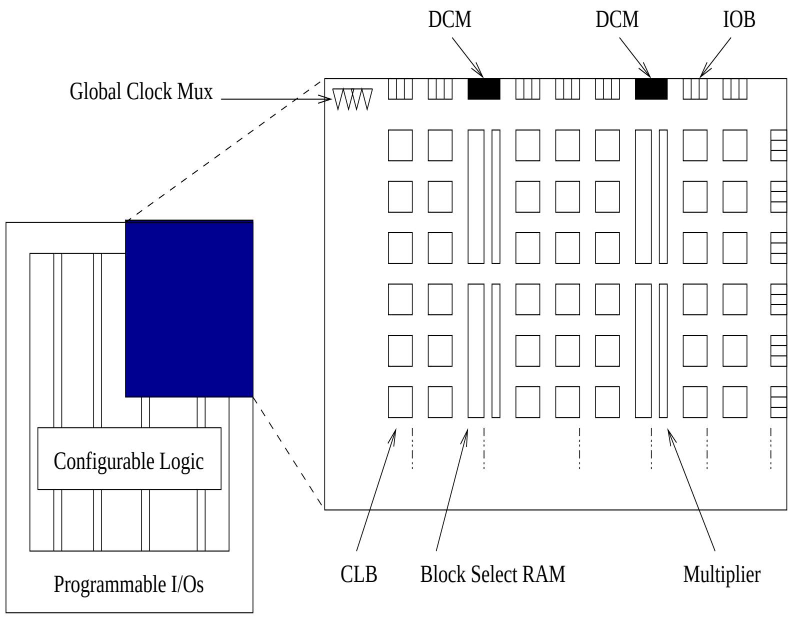

Figure 2.7: Virtex-II Architecture Overview

As can be seen in Figure 2.7, the CLBs form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip.

Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values are loaded into the memory cells on power-up, and can be reloaded if necessary to change the function of the device.

The CLBs of the Virtex-II FPGA are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix, as shown in Figure 2.8 where four slices are included. Each slice contains two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexors, and two storage elements. A single slice is shown in Figure 2.9 where each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element. The output from the function generator in each slice drives both the slice output and the D input of the storage element. Figure 2.10 shows the detailed structure of a single slice.

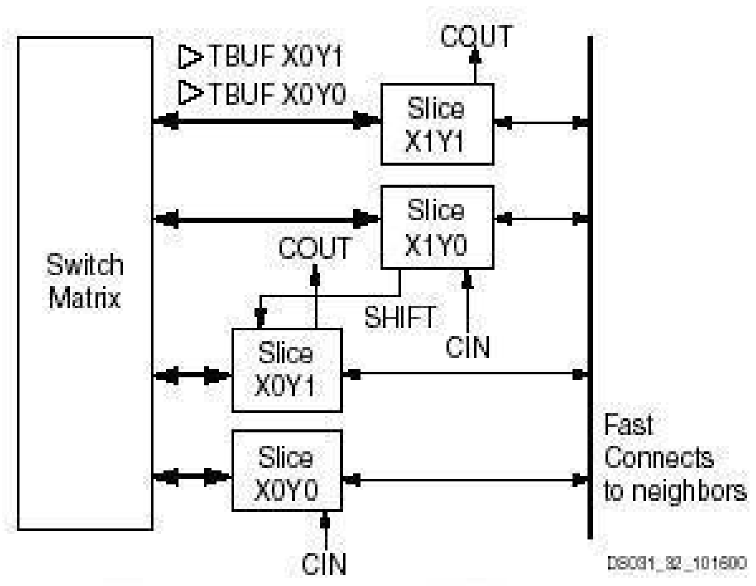

Figure 2.8: Virtex-II CLB Element

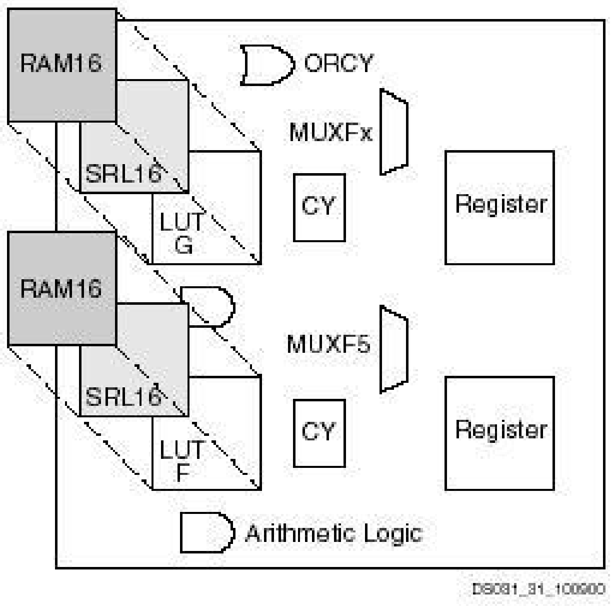

Figure 2.9: A Corase Description of A Slice

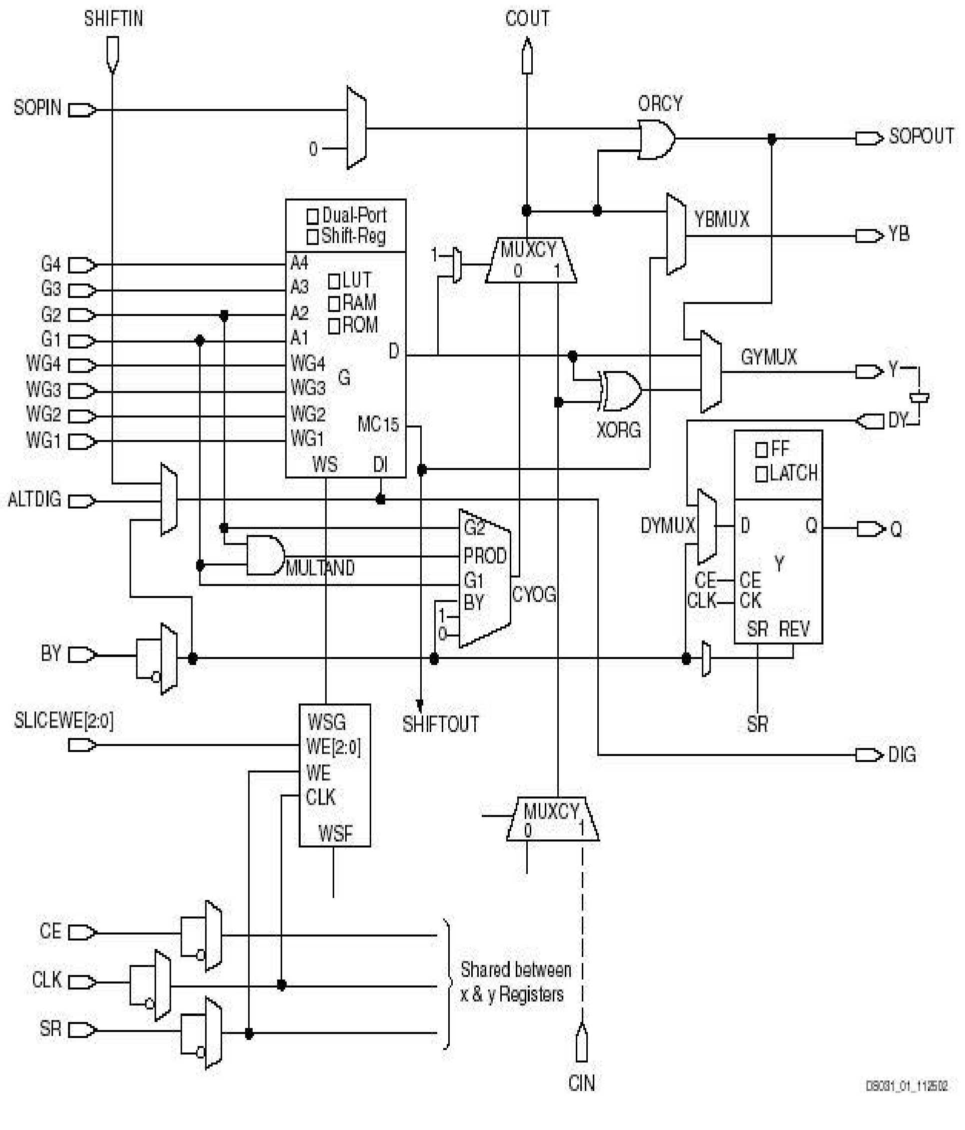

Figure 2.10: A Detailed Description of A Slice

2.2.2 Compile-time versus Run-time reconfigurations

FPGAs as a kind of re-programmable device have a wide use in RC area in which some general-purpose hardware agent is *configured* to carry out a specific task, but can be reconfigured on demand to carry out other specific tasks.

From a time domain point of view, there are two kinds of reconfigurations of FPGAs: Compile-Time Reconfiguration (CTR) and Run-Time Reconfiguration (RTR) both illustrated in Figure 2.11. As can be seen, the configuration of the re-programmable device will not change after it is running in CTR, whereas different configurations are mapped in and out on the re-programmable device after it starts. If the task to be solved can be mapped simultaneously, CTR offers better performance in terms of speed over RTR, because only one configuration time is considered. However, when the task to be loaded onto FPGAs is too complex to be loaded simultaneously, RTR is an alternative to swap different configurations in and out of the FPGAs as they are needed during the execution.

A few different configuration memory styles can be used with reconfigurable systems, as shown in Figure 2.12. A single context device is a chip in which the complete configuration has to be set by serial programming bits. Currently most of the commercial FPGAs are of this variety. To implement RTR on this kind of device, the configurations must be grouped into full contexts, and the complete contexts are swapped in and out of the device as required.

A multi-context device has multiple layers of programming bits, so that each layer can be active at a different moment. A fast-context switch is the main advantage of a multi-context FPGA over a single-context FPGA. The multi-context

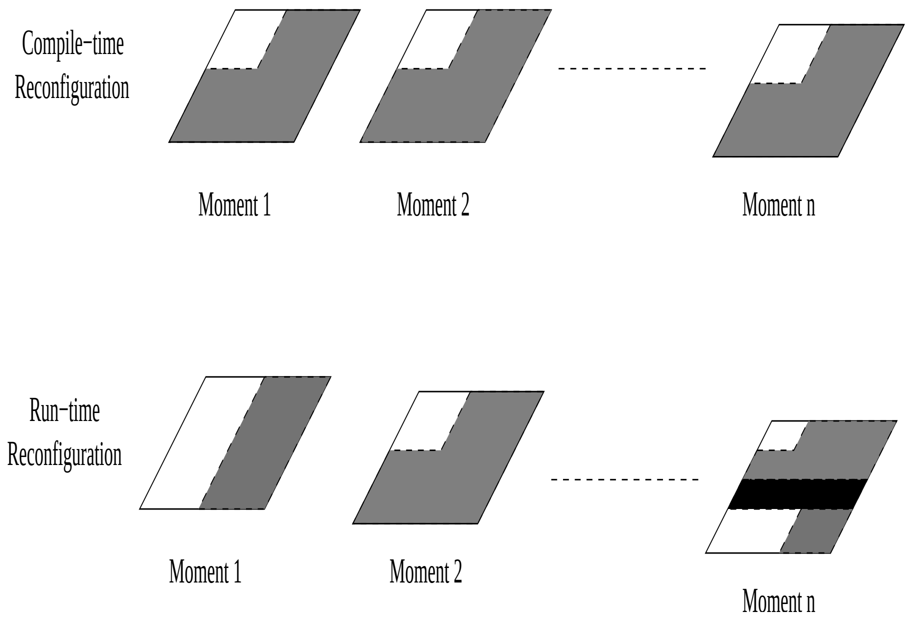

Figure 2.11: An Illustration of Compile-Time and Run-Time Reconfiguration

design allows one context to be configuring while another is in execution.

Some devices can be selectively programmed without a complete reconfiguration and are called partially reconfigurable. This kind of device is also suitable for RTR because it allows configurations which occupy only a part of the total area to be configured onto the FPGA without removing all of the configurations already present. In this way, the configuration time is less than a full-chip reconfiguration due to the reduction of data traffic.

Although RTR can offer more room for complex tasks, several issues are still under improvement. The first big issue is the compilation issue that is not encountered by CTR. Compilers must have the ability to consider the run-time reconfigurability when generating the different circuit configurations, not only to be aware of the increase in time-multiplexed capability, but also to schedule reconfigurations

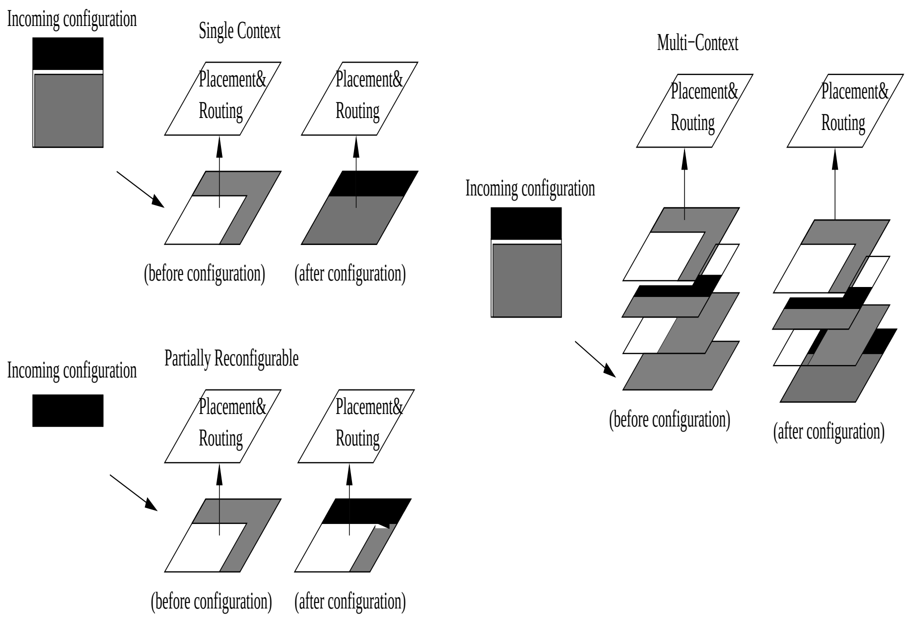

Figure 2.12: The Different Basic Models of Reconfigurable Computing: Single Context, Multi-Context and Partially Reconfigurable.

so as to minimize the configuration time overhead. The second issue is how to minimize the configuration time. Compression techniques are introduced to decrease the amount of configuration data that must be transferred to the FPGA; optimization techniques can be used to prevent unnecessary reconfiguration events from happening (i.e. a cache for reconfigurable hardware might help to provide a faster reconfiguration).

2.3 Hardware/Software Co-design

The term Hardware/Software Co-design appeared in the early 1990s. Although microprocessors had been used for over a decade at that point, microprocessor-based systems were almost board-level systems. There were designers whose work was to integrate microprocessors with standard hardware components on a board. Assembly language was widely used in this kind of system. These designers were separate from IC designers, whose work was to design Integrated Circuits (ICs) on chips.

However, people clearly realized that the microprocessor-based system design would also become important to IC designers. Moore's law [1] indicates people that IC chips would eventually be large enough to comprise both CPU and other dedicated hardware subsystems.

Hardware/Software Co-design emerged as a kind of technology to assist researchers and designers in handling/managing embedded system design. Hardware/Software Co-design can predict through analysis methods if a system meets its performance, power, and size; and synthesis methods which can be used to rapidly evaluate many potential designs.

2.3.1 Early Stages

The SOS system [SPra92] designed by Prakash and Parker was one of the earliest Co-design efforts. The system could not only synthesize an arbitrary multiprocessor topology, but also schedule and allocate processes onto the multiprocessor. The mixed integer-linear program model which was used to solve synthesis problem was slow and could not deal with large problems.

[1]The observation made in 1965 by Gordon Moore, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted that this trend would continue for the foreseeable future. In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has blessed.

The SOS system mainly dealt with multiprocessors, whereas one year later the partitioning of hardware/software surfaced as an important first step in creating models and algorithms for systems containing both CPU and dedicated hardware. The Vulcan system from Stanford [RKGu93] and the Cosyma system from the Technical University of Braunschweig [RErn93] were based on this kind of system.

Hardware/Software partitioning maps a design onto the architecture, as shown in Figure 2.13, where a CPU and one or more application-specific ICs are integrated in a system.

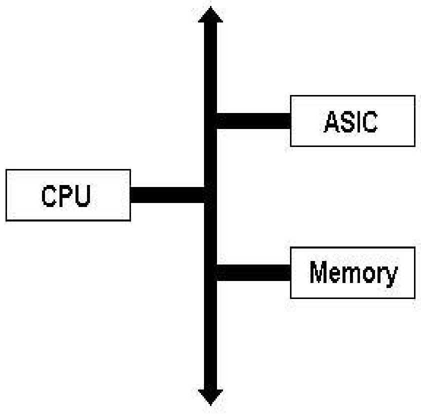

Figure 2.13: Target Architecture for Hardware/Software Partitioning

In early designs, the ASIC and CPU were generally assumed to be located on separate chips, whereas today they can be embedded into only one chip. In this kind of architecture, the computationally intensive tasks should be moved to ASIC and the rest of the work should be moved to the processor.

Both Vulcan and Cosyma captured specification of system functionality and constraints by a C-like program. The strategy of hardware/software partitioning was determined by the analysis of the performance and cost of various implementations. Vulcan started with all functionality in hardware and moved operations to CPU to minimize cost. Cosyma, on the other hand, put all functionality on the CPU and moved operations to ASIC to meet a certain performance.

The performance analysis on hardware, software and the whole system was very important in both Vulcan and Cosyma system. The goal of hardware design was to determine the maximum clock frequency. The goal of software performance analysis was to solve the problem similar to a well-known hardware problem: worst-case execution time. The analysis on the whole system was to determine a solution which meets the cost and performance requirements.

Both Cosyma and Vulcan used single-threaded computational models, where CPU sat idle while the ASIC was performing its work.

2.3.2 Current Stage

Hardware/Software Co-design quickly took off after several problems were tackled. Co-simulation is a big component of a Co-design methodology in which mixed abstraction levels are simulated to validate the design. At this time, the performance was traded off for accuracy in simulating various implementations.

Becker, Singh and Tell [DBec92] proposed a Co-simulator that linked a hardware simulator to executions of application software in 1992. Another work on simulation was introduced in the Ptolemy environment [ea94].

The worst-case execution time problem for software attracted more attention. Li, Malik and Wolfe [YT95] developed an implicit path-analysis algorithm which proved to be efficient.

The means to evaluate the system performance is yet another important issue in Co-design. Rate-monotonic scheduling [CLLi73] received attention as a method for analyzing the performance of a set of processes on a single CPU. Yen and Wolf [TY98] proposed a multiprocessor performance algorithm that analyzed the performance of a set of processes on a network of processors.

Hardware cost estimation, methods for targeting more general architectures, the low power issue, and system implementation issues such as interface generation also attracted a lot of attention in the Co-design area.

Today some FPGAs from several manufacturers contain not only programmable logic but also CPUs as well. The problem with this type of architecture is the communication between CPU and ASIC. Delays such as physical communication delay, synchronization delay, etc. can nullify any performance achieved by ASIC.

The kind of language that should be used to describe the whole system is another issue. C programming language is good for describing algorithms, but not good for specifying concurrent systems. Hardware Description Languages (HDLs) like VHDL and Verilog are geared more toward hardware.

2.3.3 Challenges in Co-design

Although some of the problems in Hardware/Software Co-design have already been solved, several long-standing problems still remain. What computational models should be used to describe hardware and software systems? How can system-level performance be analyzed precisely? What kind of memory should be used? Are there any other better architectures? How can system-level power be managed effectively? All these issues need to be tackled by researchers in the near future.

2.4 Face Recognition Methodologies

The research interest in face recognition grew tremendously during the 1990's. One of the main reasons for this growth is studies on neural network classifiers. Generally, two major approaches can be used to solve the face recognition problem: geometrical local feature-based methods, and holistic template matching-based systems. In the first approach, local features (such as eye, nose, mouth, hair, etc.) are extracted and then standard statistical pattern recognition techniques or neural network approaches are employed for matching faces. One of well-known geometrical-local feature-based methods is the Elastic Bunch Graph Matching (EBGM[Wisk99]) technique. Apart from EBGM, the whole face region is processed using the Holistic approach. In this method, the features are extracted from the whole face region. One of the methods for extracting features in a holistic system is to apply statistical methods like Principal Component Analysis (PCA). In addition to these two approaches, ANN method can also be used on the whole face image to achieve identification. The well-known algorithm in ANN is *backpropagation* algorithm, in which the gradient descent method is used to adapt neuron parameters to recognize each pattern fed into the network. The recognition methods are reviewed as statistical and neural network-based approaches respectively.

2.4.1 Statistical Methods

Basically, statistical methods include two kinds of approaches. The first approach is a template matching-based system. In this system, the training and testing face images are matched by measuring the correlation between them. Another approach is a projection-based method like Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA), etc. The second approach attempts to overcome the drawback of the first approach where the classification work takes place on the whole image which has extremely high dimensionality.

- *Template Matching:* Brunelli and Poggio [Brun93] compared a geometric feature-based technique with a template matching-based system. The simplest template matching compares the image (2-D intensity values) with a single template using a distance metric.

  Although matching raw images to achieve recognition works under limited circumstances, the largest drawback is obvious where the result is very sensitive to face orientation, size, variable lighting conditions and noise. The reason behind this phenomenon is that extremely high dimensionality is used in recognition. In order to reduce the dimensionality, some methods for feature extraction such as projection-based system are required.

- *Face Detection and Recognition by PCA*

  The Eigenface Method of Turk and Pentland [Turk91] is one of the most famous in literature, and is based on the Karhunen-Loeve expansion. In their method, face images are treated as 2-D data, and the classification of face images is achieved by projecting them on to the eigenface space which contains eigenvectors obtained by the variance of the face images. The method was applied to a database of 2500 face images of 16 subjects, digitized at all combinations of 3 head orientations, 3 head sizes and 3 lighting conditions. The experiment results show that the system is robust to illumination changes, but degrades quickly when the scale changes.

- *Face Recognition by LDA*

  Etemad and Chellappa [Etem97] proposed a method using Linear/Fisher Discriminant Analysis (LDA) for the face recognition process. The LDA of faces provides a small set of features that carries the most relevant information for classification purposes. The features are obtained through eigenvector analysis of scatter matrices with the objective of maximizing between-class variations and minimizing within-class variations. The result is an efficient projection-based feature-extraction and classification scheme. For a medium-sized database of human faces, excellent classification accuracy is achieved with the use of very-low-dimensional feature vectors.

2.4.2 Artificial Neural Network based Approach

Neural Network approaches can be used in face recognition in both a geometrical local-feature based manner and a holistic manner.

2.4.2.1 Geometrical-local feature-based ANN

Temdee et al. [PT99] proposed a frontal view face recognition method by using fractal codes which are determined by a fractal encoding method from the edge pattern of the face region (covering eyebrows, eyes and nose). The obtained fractal codes are fed as inputs to a *backpropagation* Neural Network for identifying an individual. The ORL face database is chosen to test their system. A correct recognition rate of 85% was reported.

2.4.2.2 Holistic-based ANN

Applying the *backpropagation* algorithm on the intensity values of the face image can also achieve high rate recognition [Mitc97]. The detailed explanation of this method was presented in Section 2.1. Since it was proven to be an effective method for solving facial recognition problem, determining how to implement the *backpropataion* algorithm in both a flexible and fast way is an interesting task. In this thesis, two embedded computing implementations are proposed. Results obtained show that a system containing both dedicated hardware and a soft processor achieves both flexibility and performance.

2.5 Summary

In this Chapter, the face recognition problem is introduced as the test application of MLP-BP ANNs; FPGA is the RC platform and Hardware/Software Co-design is the approach used to design systems containing both dedicated hardware and a processor. In the next Chapter, a literature survey on implementing ANNs onto FPGAs and ASICs is presented.

Chapter 3

Literature Review

A detailed comparison of existing computing platforms which can be used to implement Artificial Neural Networks (ANNs) is provided in Appendix E. General computers are flexible platforms widely used to implement algorithms. However, ANNs are inherently distributed processing elements connected together which cannot be efficiently implemented by a traditional sequential computer architecture. Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) are two platforms which are widely used when acceleration of ANNs is desired. A system containing both hardware and software (i.e., processor) is rarely used in implementing ANNs. In this Chapter, past efforts on implementing ANNs onto FPGAs and ASICs will be reviewed.

3.1 FPGA-based Artificial Neural Networks

FPGA-based reconfigurable computing architectures are well suited to implement ANNs due to their concurrency and rapid reconfigurable nature. With this nature, the weights and topologies of ANN can be easily configured onto FPGAs. Past efforts made at implementing ANNs onto FPGAs will be reviewed. This is followed by the analysis of signal representation and transfer function used in each implementation.

RRANN [Eldr94] is a run-time reconfiguration implementation of *backpropagation* algorithm using Xilinx XC3090 FPGAs. In this implementation, the *backpropagation* algorithm is divided into three sequentially executed stages: feed-forward, back-propagation and update. There is only one stage used at any given time, which makes more hardware resources available for each stage. The RRANN architecture was proven to be capable of converging on a training set and also learning important generalities about that training set. Results show 92% of approximations were within two quantization errors (1/16) of the actual value tested by realistic inputs. With random inputs, 88% of approximations were within two quantization errors. The target application of RRANN is to learn how to approximate centroids of fuzzy sets. The RRANN was a successful run-time reconfigurable ANN architecture capable of implementing non-trivial neural network algorithms by adding small amounts of hardware. Moreover, more neurons can be easily added by simply connecting another FPGA to the bus, making RRANN very scalable. In addition, the size of the network is not limited by the total number of neurons in the network but by the number of neurons in the largest layer. The improvement that might be made to RRANN to increase the resolution of some key data types so that it could converge for more applications.

Hikawa [Hika99] created another *on-chip learning* [1] Multilayer Neural Network (MNN). Instead of run-time configuration, this implementation uses compile-time configuration where all stages are implemented at the same time. A tri-state activation function yields an algorithm that replaces multiplication by a simple combination of AND and SHIFT operations. Similarly, the derivative of the activation function required for *backpropagation* is replaced by a tri-state function. Although the modifications or the simplifications to the activation function make MNN suitable for hardware implementation to 2D classifying problems, it cannot guarantee that other applications could be solved by MNN.

Beuchat *et al.* [JB98] created an FPGA platform - REconfigurable Network COmputer, called RENCO. Four Altera FLEX 10k130 FPGAs work around a standard commercial general-purpose processor (Motorola MC68en360) as a coprocessor in this system, and the target application was hand-written character recognition. Due to the existence of processor and reconfigurable hardware at the same time, RENCO can be used to implement *backpropagation* using Hardware/Software Codesign. However, the author did not mention how to partition the *backpropagation* algorithm.

Ferrucci and Martin [Ferr94, Mart94] developed a platform called Adaptive Connectionist Model Emulator (ACME) which contains multiple Xilinx XC4010 FPGAs. ACME was successfully used to learn the 2-input XOR problem [Ferr94] by

[1] *On-chip learning* occurs when the learning algorithm is implemented in hardware. *Offline learning* occurs when learning (i.e., modification of neural weights and thresholds) has already been done before the hardware system is implemented.

implementing a 3-input, 3-hidden, and 1-output network. However the learning capability of the system architecture is only verified by high-level software simulation. Another platform created by Skrbek [Skrb99] is referred to as ECX system. ECX could implement perceptron and Radial Basis Function (RBF) neural networks, and several applications such as parity problem, digit recognition, inside-outside test, and sonar signal recognition are tested on this system. Since the linear approximation is used for functions such as 2^x, log_2^x, sigmoid-like, and gauss-like functions, the precise results cannot be produced which might have a negative influence on the use of this system.

The above systems are efforts at implementing *backpropagation* algorithm on a multi-layer perceptron structure neural network. In addition to MLP-BP type, other types of ANN also exist.

Flexible Adaptable-Size Topology (FAST) was built by – Perez-Uribe [PU99b] in which three different kinds of un-supervised neural networks – adaptive resonance theory (ART), adaptive heuristic critic (AHC), and Dyna – SARSA were implemented. In the first implementation of FAST, an ART-based neural network is implemented to solve a color image segmentation problem. A 294×353 with 61-color pixel image of Van Gogh's *Sunflowers* painting is successfully segmented by four FAST neurons into four color classifications. The AHC-based neural network is implemented in the second implementation of FAST, called FAST-AHC. The *inverted pendulum problem* [PU99a] is the target problem to be solved by this system. Results obtained show that FAST-AHC couldn't generalize as well as the *backpropagation* algorithm, but it learns fast and more efficiently. The final implementation of FAST uses a Dyna-SARSA neural network [PU99a]. The target application of this system is to control a stand-alone mobile robot for obstacle avoidance.

Although FAST was the first FPGA implementation of un-supervised learning ANN, the main limitation was its application on toy problems.

De Garis *et al.* [HdG97, dG02] built a system, called CAM-Brain Machine (CBM), in which an evolutionary neural network was implemented based on *evolutionary techniques.* [2] In this system, a genetic algorithm (GA) was used to evolve a cellular automata (CA) neural network. First, the genetic algorithm's phenotype chromosome (the configuration data of each cellular automata) was initialized to dictate how the network grows. This was followed by letting the topology of a neural network module 'grow'.

CBM is the world's largest[3] evolving neural network to date, which includes 75 million neurons. It is successful in function approximation/prediction applications, such as a 3-bit comparator, a timer, and a sinusoidal function. De Garis's long-term goal was to create a modular[4] neural networks using CBM.

Besides De Garis, other researchers such as Nordstrom [Nord95], Taveniku and Linde [Tave95] also tried to implement modular neural networks onto a FPGA-based platform. Nordstrom created a system called REMAP-α, and later REMAP-β was created by Taveniku and Linde. The difference between these two systems is the size of FPGA densities used. The REMAP-α uses Xilinx XC3090 FPGAs, whereas REMAP-β uses Xilinx 4025 FPGAs.

[2]Please refer on Yao's [Yao99] work of Evolutionary Neural Network

[3]This has been confirmed by Guinness Book of World Records

[4]An in-depth survey of modular neural networks is presented in [Auda99].

3.1.1 FPGA Data Representation

The performance of an ANN is highly dependent on the range and precision of data representation used. In the *backpropagation* algorithm, the limited range and precision of a certain data representation will increase the quantization error of neuron weights and thresholds. Too large a quantization error will prevent the algorithm from being convergent. The choice of a suitable data representation for a certain application is a very important decision that has to be made.

Four kinds of data representation are discussed here: Frequency-based method, bit-stream arithmetic method, fixed-point representation, and floating-point representation. A detailed discussion of the fixed-point and floating-point issue is in Chapter 3.

1. Frequency-based method [SM00]: This is a time-dependent data representation, because it counts the number of analog spikes in a given time window. This kind of method is very popular in analog hardware implementation of ANNs.

2. Bit-stream arithmetic method [Reyn99, Hika03]: This is a method in which a stream of randomly generated bits is used to represent a real number. The probability of the number of bits that are '1' is the value of the real number. The advantage of this method is that the required multiplications can be reduced to simple logic operations. The disadvantage of the bit-stream arithmetic method is the lack of precision. In addition, the multiplication between two bit-streams is correct only if the bit-streams are not correlated.

3. Fixed-point representation [KNic02]: This representation is reported to be

the most popular representation in FPGA-based ANN architectures. This is mainly because it has acceptable range and precision for a large set of applications, while the FPGA area consumed by this representation is less than the floating-point representation.

4. Floating-point representation [KNic02]: This representation is very popular in general purpose computers to represent a real number. Until now, no successful floating-point ANN implementation has been reported. The reason is that although this kind of representation has very little quantization error, the hardware implementation of most floating-point operations is more complex than its fixed-point counterpart.

3.1.2 FPGA Transfer Function Implementation

The *backpropagation* algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution to the learning problem. Since this method requires computation of the gradient of error function at each iteration step, the continuity and differentiability of the error function must be guaranteed. That is why a transfer function must be used, and the sigmoid function is the most popular one. In FPGA implementation, two methods are popular for implementing sigmoid function. One way is to use Taylor series of sigmoid function for approximation and the other way is lookup table method.

1. Taylor series method: Galindo et al. [GHML98] proposed an implementation of the sigmoid activation function 3.1 which yielded an error of 4.48% with

respect to the continuous case. In this implementation, positive and negative numbers are treated separately. Here, INT(n) is the integer part of the number n and FRAC(n) is the fractional part (n is 2's complement binary coding).

$$a_{old}(n) = \begin{cases} \frac{1}{2} \times \frac{1}{2^{INT(n)}} \times \frac{1}{1+FRAC(n)} & n <= 0 \\ 1 - \frac{1}{2} \times \frac{1}{2^{INT(n)}} \times \frac{1}{1+FRAC(n)} & n > 0 \end{cases} \tag{3.1}$$

Inspired by Galindo's implementation, Marco et al. [httpb] proposed a better implementation generated by Taylor Series. The resulting function 3.2 gives an error of 0.51% with respect to the continuous model in the sigmoid function.

$$a_{new}(m) = \begin{cases} 0.571859 + (0.392773)m + (0.108706)m^2 + & \\ (0.014222)m^3 + (0.000734)m^4 & -\infty < m <= -1.5 \\ \frac{1}{2} + \frac{1}{4}m - \frac{1}{48}m^3 + \frac{1}{480}m^5 & -1.5 < m < 1.5 \\ 0.428141 + (0.392773)m - (0.108706)m^2 + & \\ (0.014222)m^3 - (0.000734)m^4 & 1.5 <= m < \infty \end{cases} \tag{3.2}$$

2. Lookup table method: The lookup table method is fairly easy, in which uniform samples are taken from the center of a sigmoid function and are stored in a table for lookup. The region outside the center of the sigmoid function is still approximated in a piece-wise linear fashion.

   Gilberto et al. [Cont02] implemented an activation function using a lookup table (in the form of an ROM) that approximates the sigmoid function:

$$f(a) = \frac{10}{1 + e^{\frac{-a}{8}}} \tag{3.3}$$

The sigmoid function has been approximated by taking only 16 sample values equidistant from each other. Figure 3.1 shows how this approximation is performed.

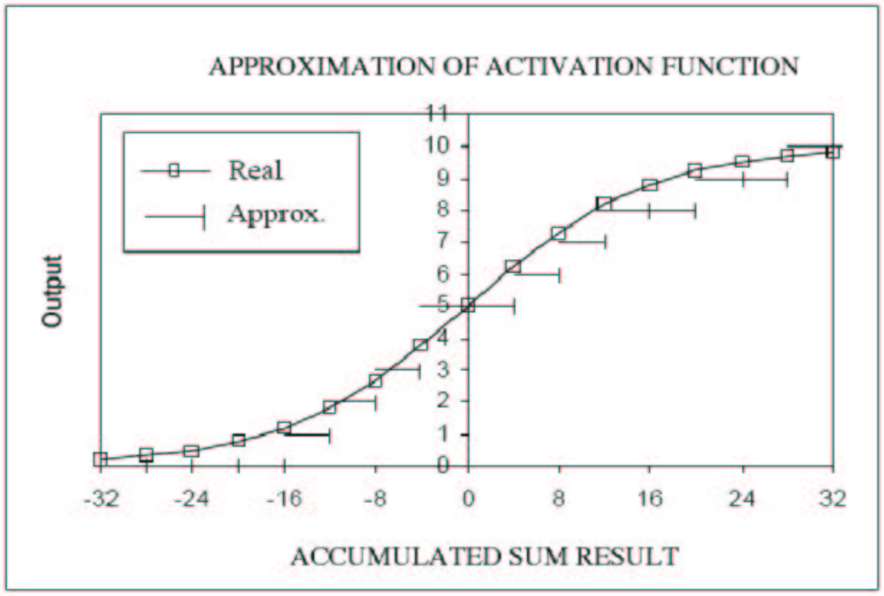

Figure 3.1: An Approximation of Sigmoid Function by Lookup Table Method

3.2 ASIC-based Backpropagation Algorithm

There is a rich history of attempts in implementing ANNs by ASIC (Application Specific Integrated Circuits). The Neurochips are built from dedicated neural ASICs. These neurochips can be digital, analog or hybrid (Figure 3.2).

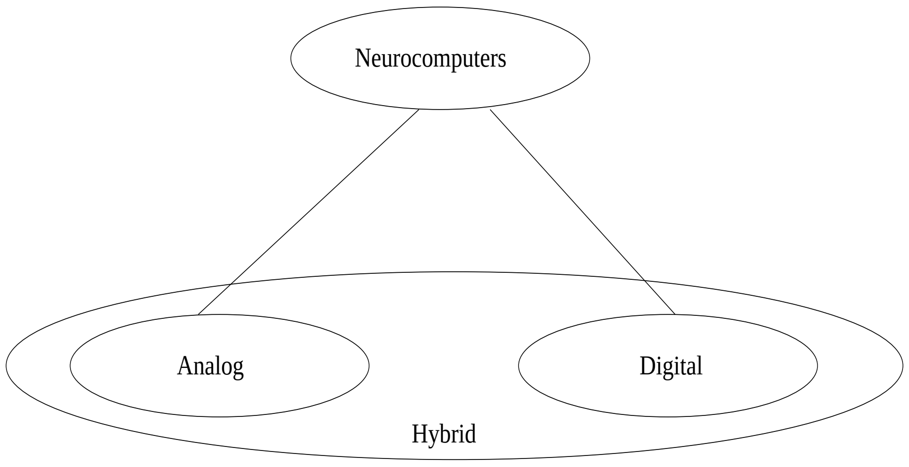

Figure 3.2: ASIC-Based Neural Network Categories

- Digital Neurochips: Digital Neural ASICs are very powerful and mature neurochips. Digital techniques offer high computational precision, and high reliability. The disadvantage is the relatively large circuit size compared to analog implementations.

  Two well-known digital Neurochips are CNAPS [McCa91] and SYNAPSE-1 [RU93]. These Neurochips were designed for a wide range of neural network algorithms. In addition to these, NESPINN (Neurocomputer for Spiking Neural Networks), designed at the Institute of Microelectronics of the Technical University of Berlin, was optimized more strictly to a certain class of neural networks: spiking neural networks. Since biological neurons use short and sudden increases in voltage to send information known as action potentials, spikes or pulses, spiking neural networks using pulse coding model neurons on a level relating more closely to biology. One NESPINN-Board is designed to compute about 10^5 programmable neurons in real-time [JA96].

- Analog Neurochips: Analog electronics have some interesting characteristics that can be used directly for neural network implementation. For example, operational amplifiers (OpAmps), are easily built from single transistors and automatically perform neuron-like functions, such as integration and sigmoid transfer. The advantage of analog neurochips is their high speed. The disadvantages are the susceptibility to noise and process-parameter variations that limit computational precision. Another problem is the representation of adaptable weights which limit the applicability of analog circuits. Weights can be represented by resistors, but these are not adaptable after the production of the chips. Capacitors, floating gate transistors, charge couple devices (CCDs), etc. allow for adaptable weights. However, the problems with these techniques arise from process-parameter variations across the chip, limited storage times, and lack of compatibility with standard VLSI processing technology. Although analog chips will never reach the flexibility attainable with digital chips, their speed makes them very attractive. Some successful analog chips are Intel ETANN [H90] and [MY93] [H94].

- Hybrid Neurochips: Hybrid Neurochips tend to combine both digital and analog technique in implementing ANN systems. The Epsilon [CS92] chip is a hybrid neurochip that uses pulse coding techniques. The Epsilon chip consists of 30 nodes and 3600 synaptic weights. With this chip, it is possible to implement robust and reliable networks using the pulse stream technique.

3.3 Summary

This Chapter reviewed previous efforts in implementing ANNs onto FPGAs and ASICs. These two kinds of platforms are suitable for implementing ANNs due to the parallelism achieved which is a good match for the inherent distributed structure of ANNs. The next Chapter focuses on the data representation used to implement ANNs on FPGAs. Two arithmetic formats are compared in terms of precision range and FPGA area for implementing ANNs on FPGAs.

Chapter 4

MLP-BP Implementations: Arithmetic Formats

4.1 Introduction

An important consideration for implementing an ANN on FPGAs is the arithmetic representation format. The problem is to achieve a balance between the need for numeric precision, which is important for network accuracy and speed of convergence, and the cost of logic areas (i.e. FPGA resources) associated with increased precision. While standard floating-point representations (i.e 32 and 64 IEEE floating point formats) offer adequate precision, they require more FPGA resources than other area-efficient arithmetic representations, such as less precise floating and fixed point formats.

This Chapter explores the *area vs. precision* design trade-off by examining the implementation of basic components of an artificial neural network using various fixed point and floating point formats. We discuss the required resources for each implementation and the impact on implementing ANN on FPGAs. This Chapter is organized as follows. Section 4.2 covers background materials regarding floating-point and fixed-point arithmetic format and multi-layer perceptron artificial neural networks. Section 4.3 presents previous work in approaching the precision vs. area trade-off. The implementation of different floating-point and fixed-point adders and multipliers is discussed in Section 4.4. This is followed by Section 4.5 which compares area consumption of floating-point and fixed-point adders and multipliers in an FPGA.

4.2 Background

4.2.1 Floating-point and Fixed-point Format

- **Floating-point format:** In general, a floating-point number is represented as $\pm d.dd...d \times \beta^e$. More precisely $\pm d_0.d_1d_2...d_{p-1} \times \beta^e$ represents the number

$$\pm(d_0 + d_1\beta^{-1} + ... + d_{p-1}\beta^{-(p-1)})\beta^e, (0 \leq d_i < \beta) \tag{4.1}$$

  Where β is called the *base* (which is always assumed to be even) and e is called the *exponent* and p is the precision. For example, if $\beta = 10$ and $p = 3$ then the number 0.1 is represented as $1.00 \times 10^{(-1)}$. If $\beta = 2$ and $p = 24$, then the decimal number 0.1 cannot be represented exactly but can be approximately represented by $1.10011001100110011001101 \times 2^{(-4)}$. The exponent is said to have a *biased* representation when the value of the exponent is

$$e = k - (\beta^{m-1} - 1) \tag{4.2}$$

Where k is the value of the exponent bits interpreted as an unsigned integer and m is the number of bits in the exponent. The floating-point number is said to be a normalized number when d_0 is '1' in equation 4.1.

One of the most common floating point formats is the IEEE 754-1985 format[ANSI85]. In this format, for single precision numbers, $\beta = 2$, $p = 24$, $m = 8$, and $e = k - 127$. The value of the floating point number can then be obtained from equation 4.1 as

$$v = -1^s(1.f)2^e \tag{4.3}$$

Where f is called the significant (a.k.a. mantissa) and equal to $p - 1 = 23$. The bit representation of this format is illustrated in Figure 4.1. The first bit is a *sign* bit where $(-1)^s$determines whether it is a positive number or a negative number.

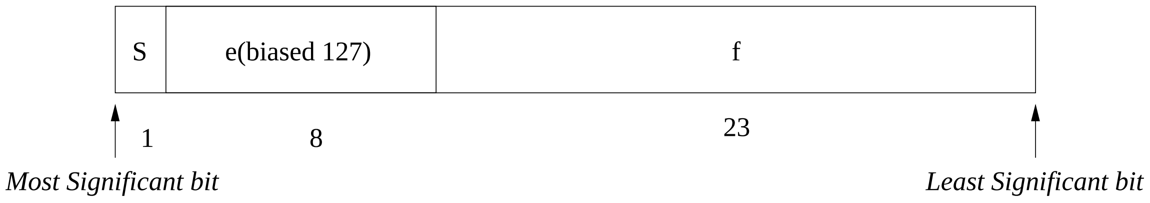

Figure 4.1: IEEE Standard 754-1985 Format

- **Fixed-point Format:**

The representation of fixed-point is illustrated in Figure 4.2. There are two parts in a fixed-point number. One is the integer part which is b_{ws-1} to b_4 and the other is the fractional part which is b_3 to b_0 as illustrated in Figure 4.2. If the base of this fixed-point number is β and it is a positive number, the decimal equivalent value can be calculated by:

$$b_{ws-1}\beta^{ws-5} + ... + b_4 + b_3\beta^{-1} + b_2\beta^{-2} + b_3\beta^{-3} + b_4\beta^{-4} \tag{4.4}$$

If the base of fixed-point number is 2, the value is determined by the type of representation used (generally 2's complement is used).

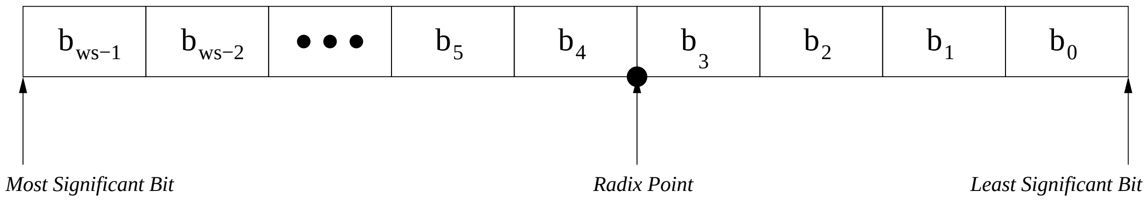

Figure 4.2: Format of a Fixed-point Number

4.2.2 Analysis of Precision and Range

To understand the difference between floating-point and fixed-point representations, we will examine an example using a 4-bit format. In order to simplify the comparison, negative numbers are avoided therefore the sign bit is not included. Figure 4.3 shows a 4-bit floating-point and 4-bit fixed-point number.

- **Floating-point:** The smallest number in this representation is decimal 0.5 ($(1.00)_2 \times 2^{-1}$) and the largest number is decimal 7 ($(1.11)_2 \times 2^2$). Precision

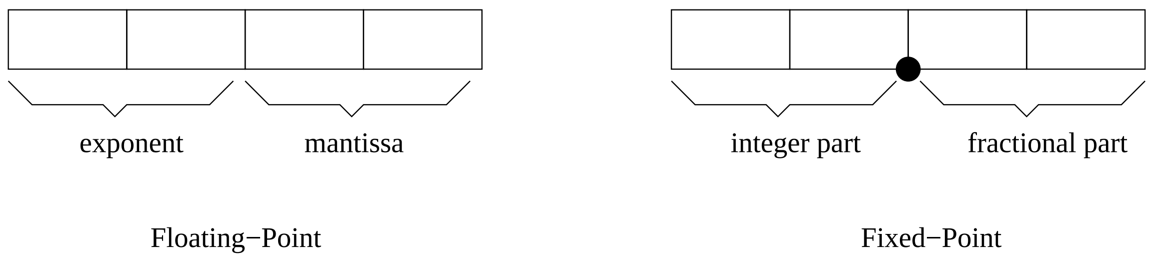

Figure 4.3: Config#1: 4-bit Floating-point vs. Fixed-point

for numbers < 1 is 0.125 while precision for numbers > 4 is 1. There are 16 normalized numbers in this representation as shown in Figure 4.4

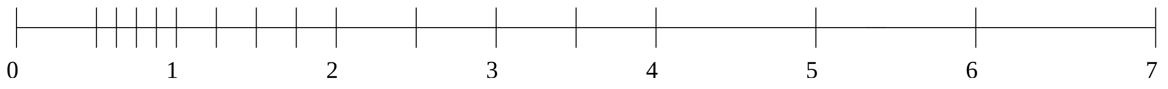

Figure 4.4: Normalized Numbers ($\beta = 2$,$p = 3$,$e_{min} = -1$,$e_{max} = 2$)

- **Fixed-point:** The smallest number is $(0.00)_2$ which is decimal 0 and the largest number is $(11.11)_2$ which is decimal 3.75. Precision is constant at 0.25. There are 16 numbers in this representation as shown in Figure 4.5.

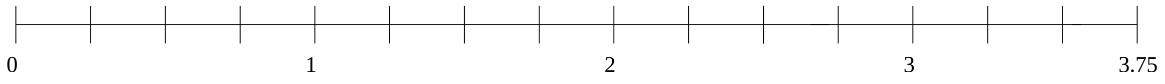

Figure 4.5: 4-bit Fixed-point Representation (fractional part:2 bits)

From the above discussion, it can be seen that the floating-point format has a large dynamic range with varied precision while the fixed-point format is limited in range but its precision is independent of the number represented. It should be noted that the range/precision is also dependent on the position of the radix point in the format. For example Figure 4.6 shows a different fixed point format using the same 4-bit representation. The numbers represented are shown in Figure 4.7 where the range is 0 - 1.875 and the precision is 0.125. As such, for a constant number of bits, increasing the range will lead to a decrease in precision and vice versa. This conclusion is valid for both floating and fixed format.

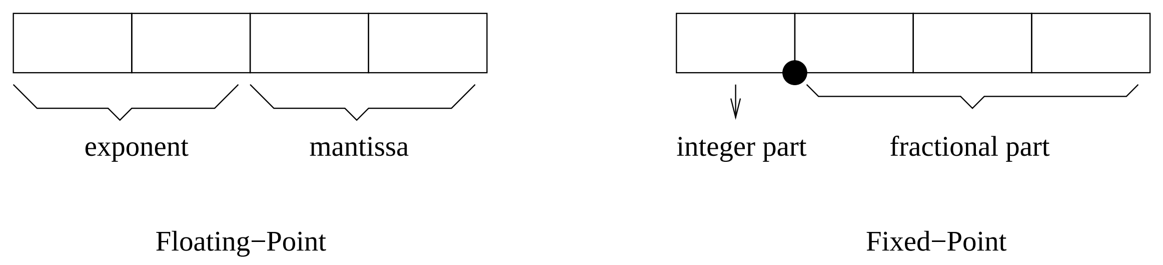

Figure 4.6: Config#2: 4-bit Floating-point vs. Fixed-point

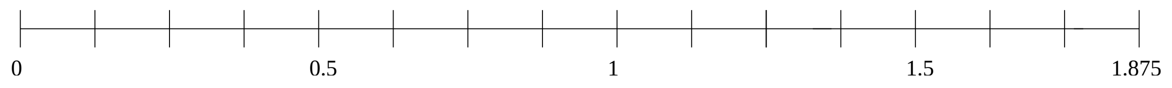

Figure 4.7: 4-bit Fixed-point Representation (fractional part:3 bits)

4.3 Ideal Data Format for MLP-BP Networks

A careful analysis of the MLP-BP network operation as summarized in Section 2.1.2 of Chapter 2 shows the following:

- The basic arithmetic operations in the forward and backward passes consist of a multiplication and a summation.
- Weights start around zero and increase or decrease as learning progresses. However, large weights lead to a network 'saturation' where only a small

subset of the network's weights are changed as illustrated by equation 2.10 and 2.22. Large weights also lead to poor generalization after training.

- Transfer functions (squashing functions) must be bounded. For example the logistic sigmoid function shown in subsection 2.1.2 maps any number > 8 to a number 1 and maps any number $<$ -8 to zero.

- As learning progresses, the error between the target and output values becomes smaller (equation 2.10 and 2.22). If the error becomes zero, all learning stops. Thus it is critical that the arithmetic format precision allows for very small error numbers to be represented.

From the above discussion, one can conclude that the ideal format for MLP-BP networks will have a high precision. Range can be limited as long as inputs and outputs are normalized since weights should be limited for learning progress. But since the size of an FPGA-based MLP-BP is proportional to a multiplier used [1], we are interested in finding the *minimum allowable precision* and *minimum allowable range* which minimize hardware area usage while not affecting ANN convergence or performance.

For MLP using the BP algorithm, Holt and Baker [Holt91] showed using simulations and theoretical analysis that *16-bit fixed-point* (1 bit sign, 3 bit left and 12 bit right of the radix point) was the minimum allowable range-precision assuming that both input and output were normalized between [0,1] and a sigmoid transfer function was used. Ligon III *et al.* [Ligo98] have also shown the density

[1] An MLP-BP is constructed by connecting processing elements to perform a certain function. Each processing element includes at least one adder and one multiplier, so the FPGA area of an MLP-BP is mainly proportional to a multiplier.

advantage of fixed-point over floating-point for older generation Xilinx 4020E FPGAs, by showing that the space/time requirements for 32-bit fixed-point adders and multipliers were less than those of their 32-bit floating-point countparts. This chapter's goal is to provide a comprehensive study of the impact of various floating-point and fixed-point formats on hardware resources taking into consideration the performance requirements discussed above and current FPGAs architectures (e.g. Xilinx Virtex-II with embedded multipliers).

4.4 RC Implementation of Floating-point and Fixed-point Adder and Multiplier

The implementations of a floating-point adder and multiplier are introduced in Pavle [Bela02]. A library of fully parameterized hardware modules for floating-point format control, arithmetic operators and conversion to/from any fixed-point format are also presented in [Bela02]. The implementation of the fixed-point adder are based on a Ripple Carry and Carry Look Ahead adder architecture. The fixed-point multiplier is implemented in two forms: parallel and serial.

4.4.1 Implementation of Floating-point Adder

Addition is one of the most complex operations in floating-point arithmetic. The algorithm for addition includes four steps:

- Re-arrangement of input operands (larger operand is directed to input 1 (**swap**)),

- Pre-shift for mantissa alignment (**shift_adjust**),
- Mantissa addition/subtraction (**add_sub**), and
- Post-shift of mantissa and increment of exponent for result correction (**correction**).

Each step of the algorithm is implemented by a dedicated module. The four sub-modules are assembled into the overall **fp_add** module as shown in Figure 4.8.

The **swap** submodule compares the exponent and mantissa fields of the input variables. Based on these two comparisons, the two floating-point inputs are appropriately directed to the output. If the exponent field of input "A" is larger, or the exponent field are equal and the mantissa of input "A" is larger, then input "A" is multiplexed to output **large** and input "B" to output **small**. Otherwise, the reverse mapping of inputs to outputs occurs. Submodule **shift_adjust** is responsible for aligning the mantissas of the larger and smaller operands. This is achieved by shifting the small mantissa to the right as many bits as the difference between the exponents. This module is also responsible for introducing the guard bit into the smaller operand's mantissa. Guard bits are introduced in the addition algorithm to provide rounding of the result. Hence, the mantissa of the sum is one bit wider than the mantissa of the inputs. Expansion of the mantissa fields occurs during alignment of the mantissa. This allows the extra information (the guard bit carries) to be saved when the smaller operand's mantissa is shifted right but as a result some least significant bits may be lost. The guard bit is introduced into the larger operand's mantissa to the right of the least significant bit and always has value '0'. Once the mantissas are aligned, it is necessary to either add or subtract them, depending on the signs of the two operands. If the signs are the same, the addition operation is constructive and the mantissas are added. If the signs are opposite, the addition operation is destructive and the mantissas are subtracted. Sub-module **add_sub** will perform this variable operation under the control of the **op** input, which is fed with the **XOR** of the input sign bits. Outputs of the overall addition algorithm are controlled by the **correction** module. If an exception is indicated on the input, the exception is propagated to the output and the result output is set to all zeros. Otherwise, if the input values are detected to be of the same magnitude, but opposite sign. The result output is blanked out to indicate zero value, as $A + (-A) = 0$. Otherwise, if an overflow in the addition of the mantissas is detected, the result mantissa is shifted to the right by one bit, truncating the least significant bit, and the most significant bit is filled with '1'. Also, the exponent field is incremented by 1, to reflect the shift in the mantissa. Finally, the floating-point value assembled from the sign, exponent and mantissa fields is presented on the output. Module **fp_add** is parameterized by the width of the exponent and mantissa fields of the floating-point format it operates on. In order to have the same criterion, the output result is truncated to have the same precision and dynamic range with the input operands.

4.4.2 Implementation of Floating-point Multiplier

In floating-point arithmetic, multiplication is a relatively straight-forward operation compared to addition. This is again due to the sign-magnitude nature of the floating-point format as illustrated in the following example:

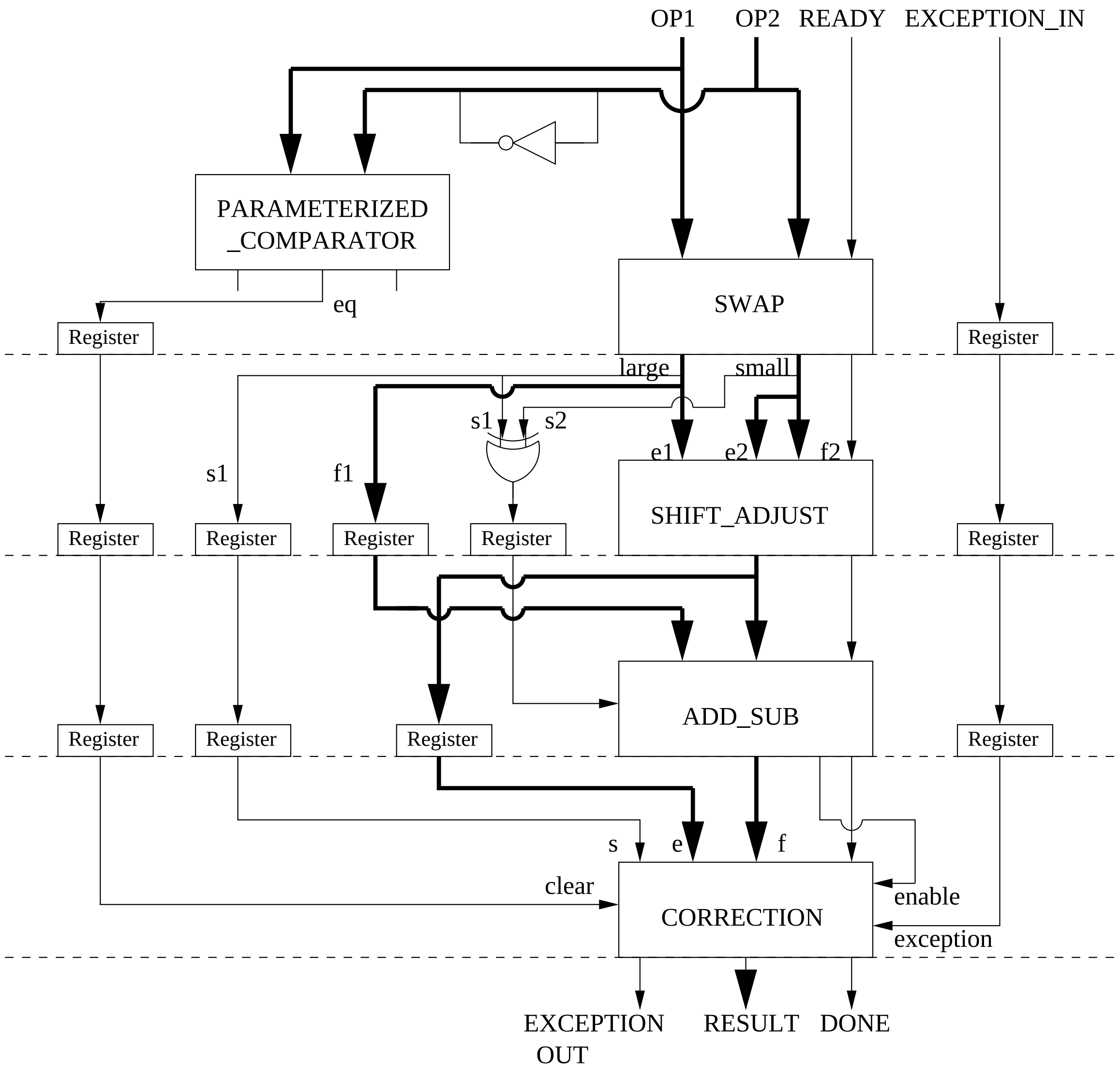

Figure 4.8: Pipelined Floating-point Adder

$$\begin{aligned}&((-1)^{s_1} \times m_1 \times 2^{e_1}) \times ((-1)^{s_2} \times m_2 \times 2^{e_2})\\ &\quad = (-1)^{\mathbf{s_1 \oplus s_2}} \times (\mathbf{m_1} \times \mathbf{m_2}) \times 2^{(\mathbf{e_1+e_2})}\end{aligned} \tag{4.5}$$

It is clear from the above that the three fields of the floating-point format do not interact with each other during multiplication and thus can be processed in parallel. The sign of the product is given as the exclusive OR (XOR) of the input value mantissas, while the exponents of the input values are added to give the exponent of the product.

The only further complication of the floating-point multiplication algorithm is the fact that the exponent field is biased. When two biased exponent fields are added, the result contains the bias twice, one of which must be subtracted. If, using IEEE standard 754 notation, E is an unbiased exponent and e is a biased exponent, it stands that:

$$\begin{aligned}e_1 + e_2 =& \\ (E_1 + BIAS) + (E_2 + BIAS) =& \\ (E_1 + E_2) + 2 \times BIAS& \\ E_p + 2 \times BIAS& \\ e_p + BIAS&\end{aligned} \tag{4.6}$$

The structure of the floating-point multiplier is presented in Figure 4.9. The **fp_mul** module is parameterized by the bit-widths of the exponent and mantissa fields of the floating-point format it processes. The bit-width of the product is $1+exp\_bits + (2\times\ man\_bits)$, where the mantissa field has twice the bit-width of the input mantissas. In order to have the same criterion, the result is truncated to have the same precision and dynamic range with the input operands.

4.4.3 Implementation of a Fixed-point Adder:

fixed-point adders can be implemented either as a Ripple Carry Adder or a Carry Lookahead Adder. According to the representation introduced earlier, a fixed-point number contains three parts: integer part, fractional part and a radix point. Since the radix is mainly used to indicate the position of both the integer and fractional part, an adder of total length of both integer and fractional part can be used to add two fixed-point numbers. The top view of a parameterized fixed-point adder is shown in Figure 4.10. Detailed implementation of the fixed-point adder based on a Ripple Carry Adder and a Carry Lookahead architecture are shown in Figure 4.11 and Figure 4.12 respectively. Table 4.1 gives an example of this parameterized adder. The term "parameterized" indicates that the length of the integer/fractional parts can be set through two generic variables: *int_lens* and *frac_lens* respectively. In this example, int_lens is set to 3 and the value of frac_lens is set to 2. The two input numbers are "10111" and "11001" (which represent decimal -2.25 and -1.75). The sum (taking Carry into account) is 110000 (which represents decimal -4) and the radix is in between the third zero and second zero from the right.

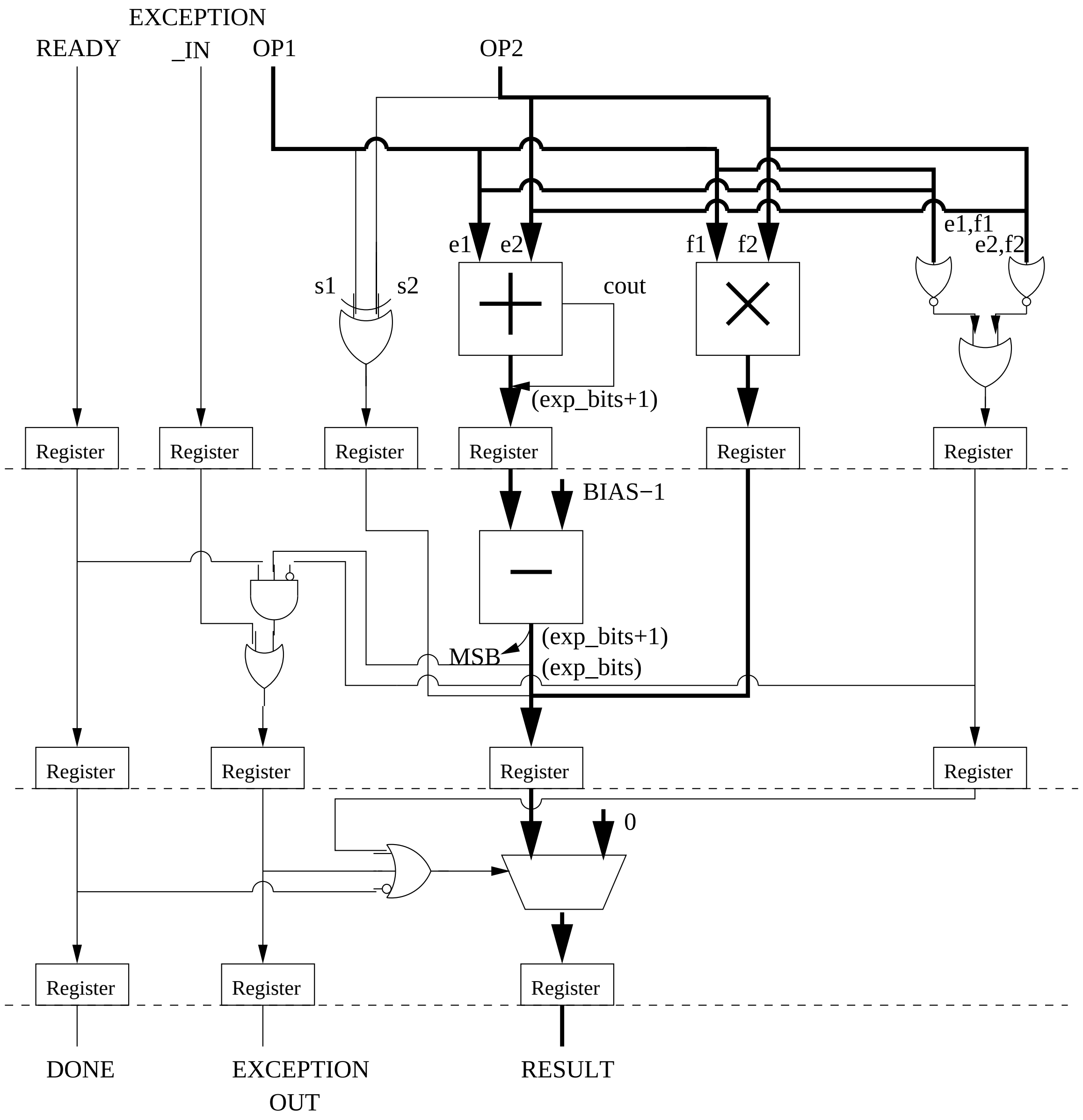

Figure 4.9: A Pipelined Floating-point Multiplier

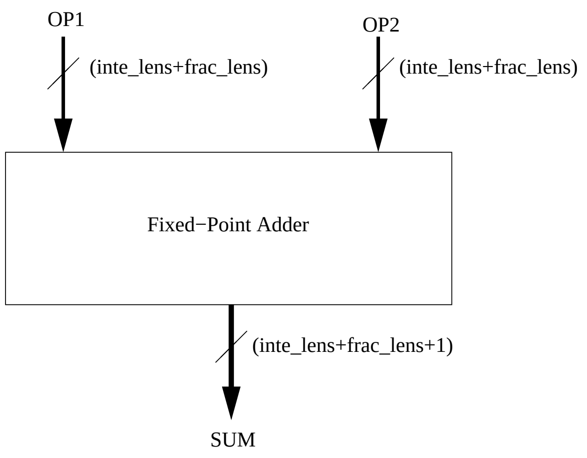

Figure 4.10: The Top View of Fixed-point Adder

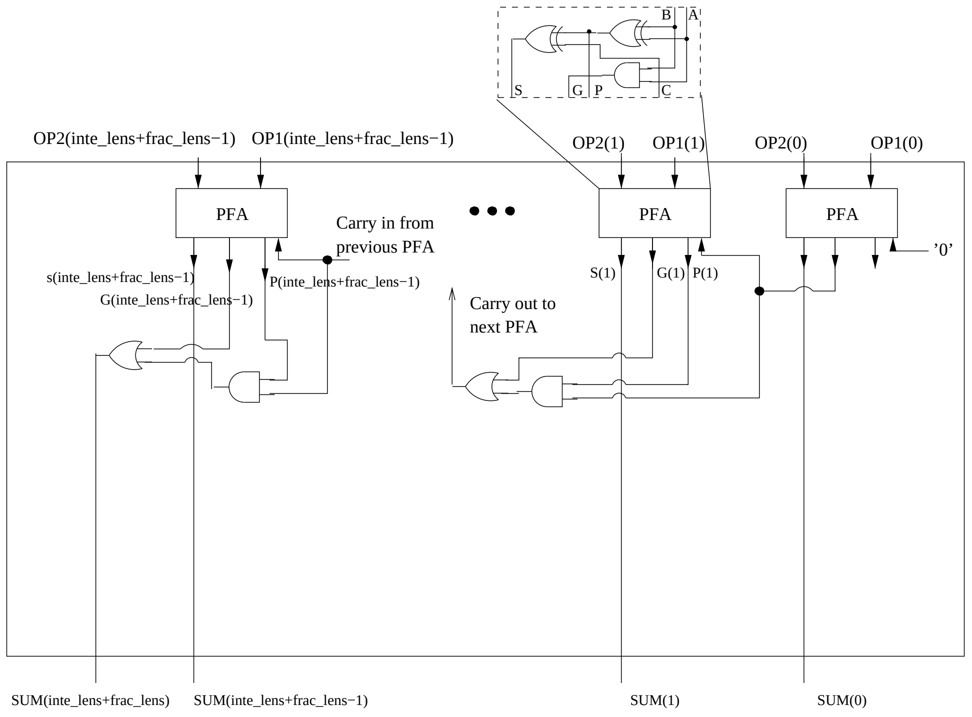

Figure 4.11: A Ripple Carry Adder Inside the Top View of Fixed-point Adder

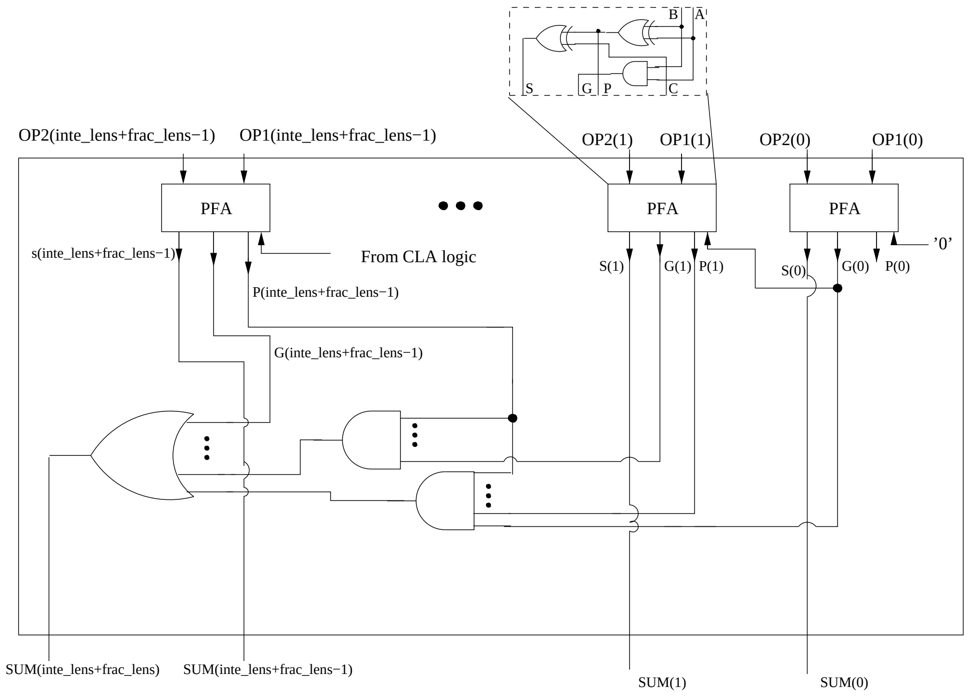

Figure 4.12: A Carry Lookahead Adder Inside the Top View of Fixed-point Adder

Generic	int_lens=3,frac_lens=2
Operation	Addition
Input data	OP1=10111,OP2=11001
Carry in	‘0’
Result	10000
Carry out	‘1’

Table 4.1: An Example of Fixed-point Addition

4.4.4 Implementation of a Fixed-point Multiplier:

As indicated earlier, pipelined fixed-point multipliers (see Figure 4.13) are more complicated than fixed-point adders. Inputs to the multiplier are signed numbers in 2’s complement representation. A requirement for fixed-point multiplication is to multiply the magnitudes of the two numbers first and then force the result to 2’s complement module if the sign of two input fixed-point numbers are different.

For unsigned multiplications, several algorithms can be used. Two straightforward types of unsigned multipliers are considered here. The first is a Serial multiplier shown in Figure 4.14 that executes in ‘n’ clock cycles (where n is the length of A and B). An example of unsigned serial fixed-point multiplication is illustrated in Table 4.2. In this example, “A” is “01111” and “B” is “00101”. The multiplier is also parameterized where *int_lens* is set to 3 and *frac_lens* is set to 2. The processing of these two fixed-point numbers can be achieved in 5 clock cycles. In the first cycle, “A” is extended to 10 bits. In the following cycles, the previous partial result is shifted left by one bit per cycle. After 5 cycles, partial results or zeros are added depending on whether the corresponding bit value in “B” is ‘1’ or ‘0’. The main advantage of Serial multipliers is the reduction in area but at the expense of lower speed.

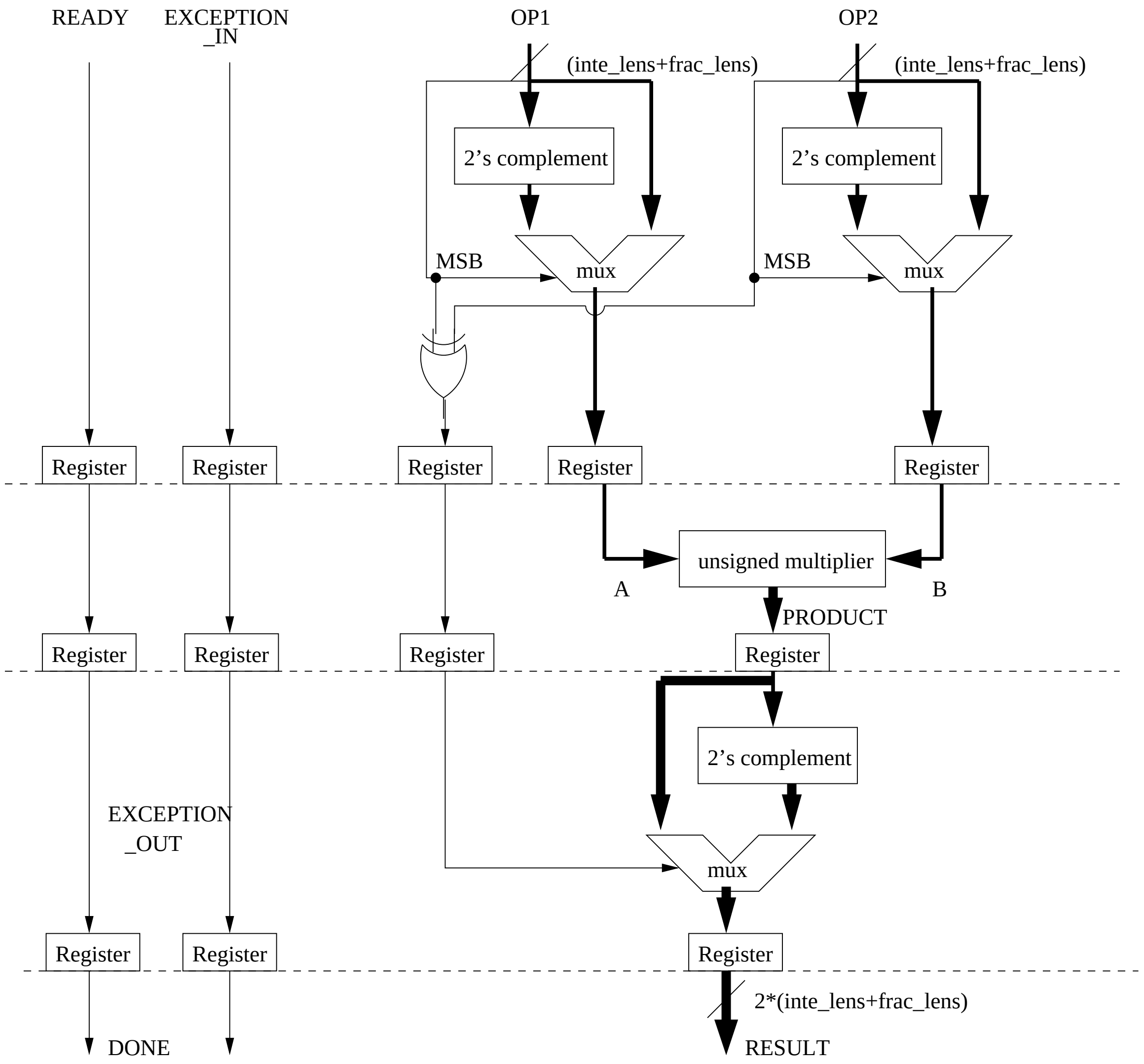

Figure 4.13: Implementation of a Pipelined Fixed-point Multiplier

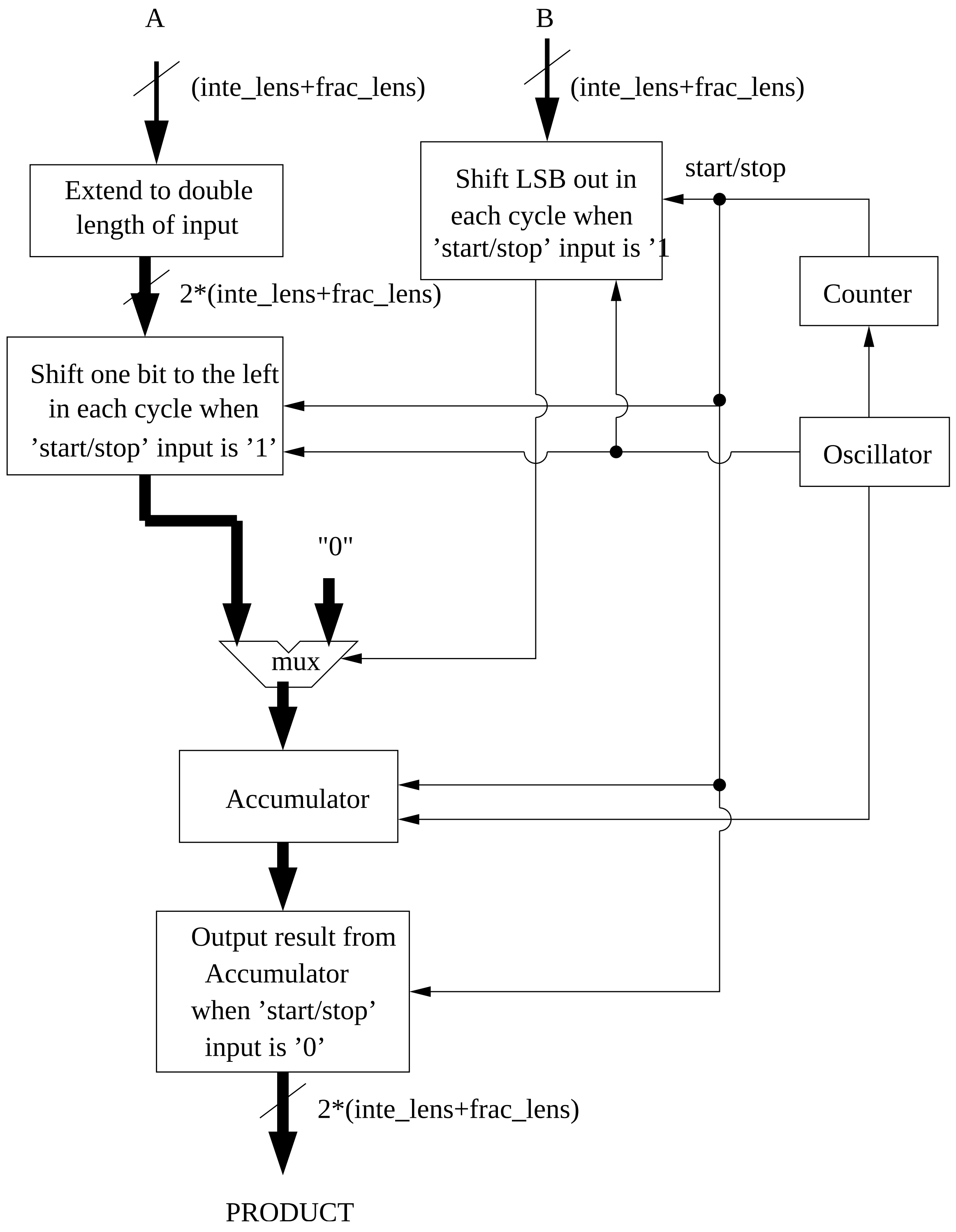
A
B
(inte_lens+frac_lens)
(inte_lens+frac_lens)
Extend to double length of input
Shift LSB out in each cycle when 'start/stop' input is '1
start/stop
Counter
2*(inte_lens+frac_lens)
Shift one bit to the left in each cycle when 'start/stop' input is '1'
Oscillator
"0"
mux
Accumulator
Output result from Accumulator when 'start/stop' input is '0'
2*(inte_lens+frac_lens)
PRODUCT

Figure 4.14: Unsigned Serial Multiplier

Cycle No.	Partial result
1	0000001111
2	0000011110
3	0000111100
4	0001111000
5	0011110000

Table 4.2: An example of Unsigned Serial Multiplication

Figure 4.15 shows the structure of an unsigned parallel multiplier. In this case, all partial results (output of multiplexers) are added at the same time. The main advantage of the parallel multiplier is speed but it requires more area than the Serial counterpart.

4.5 Results and Discussion

Different floating and fixed point formats can have different range and precision even if their total number of bits are the same. Therefore we tested 25 different formats that fall within the acceptable range and precision required for MLP-BP implementation as discussed before. For each format, a multiplier and an adder were implemented. The equivalent gate count was obtained using Xilinx ISE 6.1 (SP3) CAD tool assuming the target device is either Xilinx Spartan2E xc2s200e pq208 or Virtex-II XC2V500 FPGA. The speed grade is -6 and design flow is XST VHDL. Spartan2E contains 2,352 slices and 4,704 4-LUTs while the Virtex-II XC2V500 contains 3,072 slices and 6,114 4-LUTs.

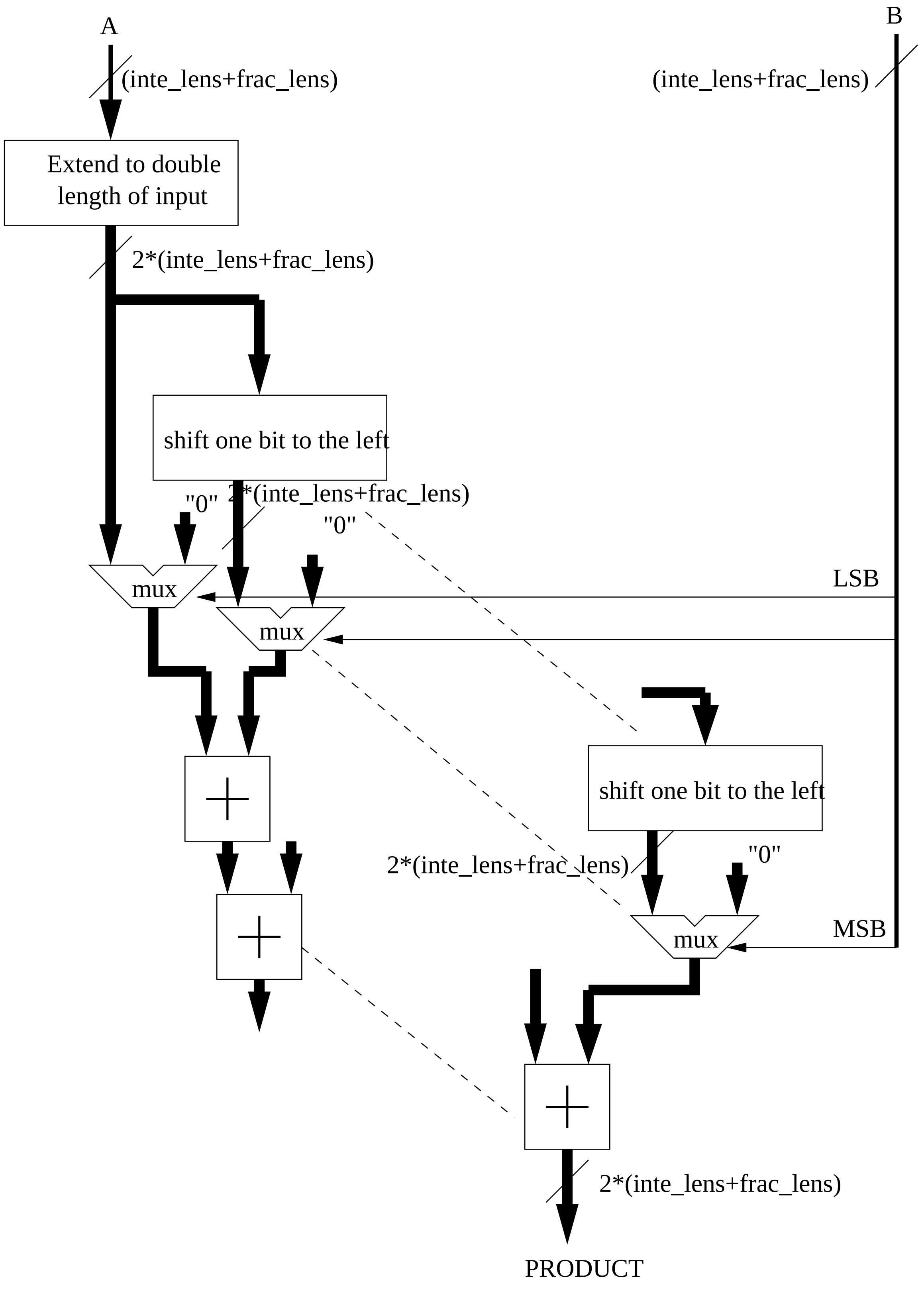

Figure 4.15: Unsigned Parallel Multiplier

4.5.1 Effect of target device

Different target devices can have significant effect on the equivalent gates results.

Attribute	**Spartan-IIE**	**Virtex-II**
Slices	2 per CLB	4 per CLB
LUTs	4 per CLB	8 per CLB
Flip-Flops	4 D-FF	8 D-FF
MULT_ANDs	4	8
Carry-Chains	2	2
SOP Chains	none	2
SelectRAM	64 bits	128 bits
Shift-Reg	none	128 bits
TBUF	2	2
Dedicated 18×18 Multipliers	none	4-168

Table 4.3: Logic Resources of Spartan-IIE and Virtex-II FPGAs

Table 4.3 shows the main attributes available within a Spartan-IIE and Virtex-II FPGA. It is clear from the Table that the main difference between the two FPGA devices is in the dedicated multipliers and Sum of Products (SOP) Chains that are used to implement large AND gates or other combinational logic functions. The effect of these differences within our format ranges is illustrated in Figure 4.16 and 4.17 where a comparison in terms of *Gate Count* between Spartan2E xc2s200e and Virtex-II XC2V500 FPGAs is shown. In Figure 4.16, a floating point adder for different formats ranging from 8-64 bits targeting both devices was implemented. The same was done in Figure 4.17 where a floating point multiplier was implemented. In both cases, the exponent part and mantissa part occupy half of the total length.

As seen from these Figures, there is little difference between the two devices for implementing adders while there is a clear difference in favor of the Spartan chip

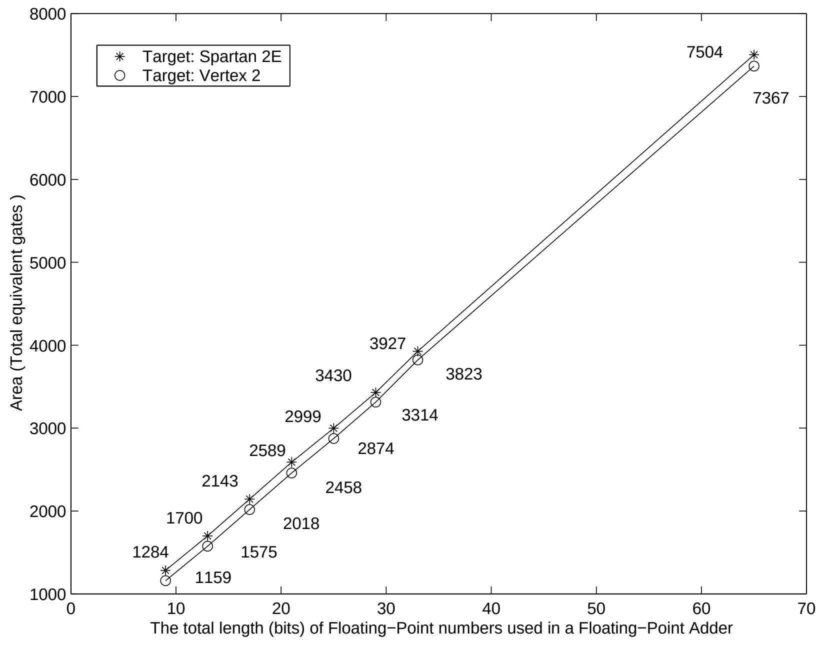

Figure 4.16: Spartan-IIE vs. Virtex-II Implementation of the Floating-point Adder

when it comes to implementing multipliers. A possible explanation behind this is due to the fact that the Virtex-II chip has 18 bits*18 bits dedicated multipliers. Each multiplier is synthesized as 4,000 gates even when both of the inputs to this multiplier are less than 18 bits (even if the inputs are just 2 bits, it is still synthesized as 4,000 gates). For Spartan-2E FPGA, the multiplication is realized by LUTs. As such in Figure 4.17 Virtex-II FPGA implementation starts from a high gate count whereas Spartan-2E FPGA starts from a much lower gate count. The gate count for the Virtex-II increases gradually since the exponent portion requires more gates for adders. The impact of these observations will be clear in subsequent sections.

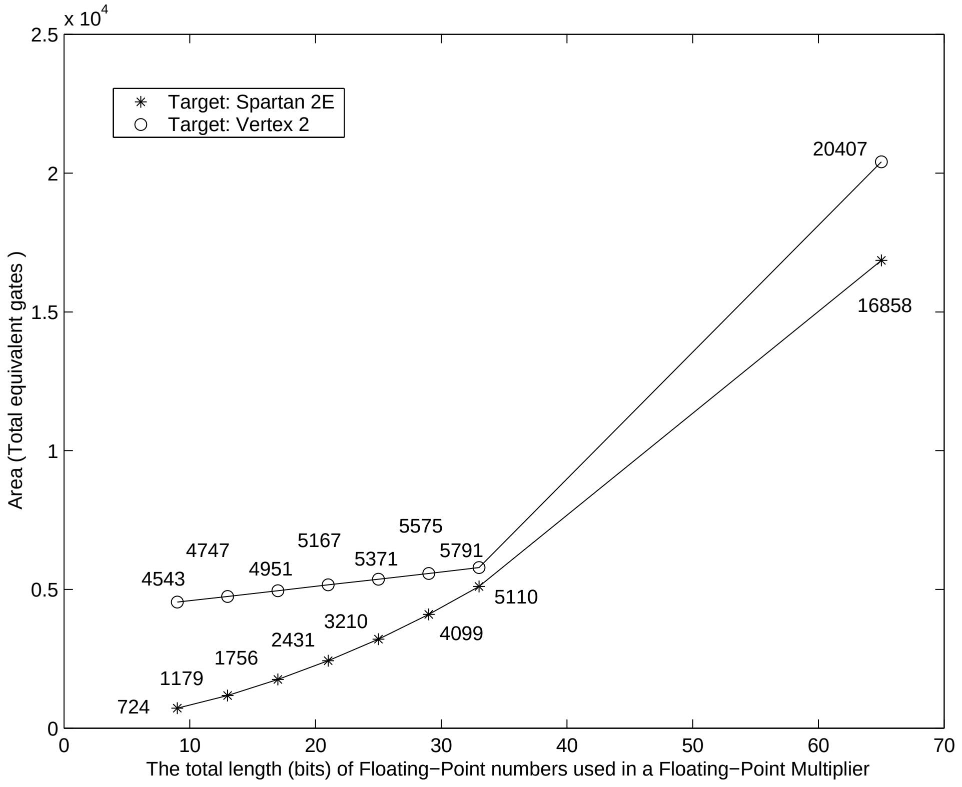

Figure 4.17: Spartan-IIE vs. Virtex-II Implementation of the Floating-point Multiplier

4.5.2 Tested formats and implementation details

The formats chosen for comparison are illustrated in Table 4.4 and Table 4.5 for fixed and floating point formats respectively. They have integer ranges [-32, +32]. These ranges are quite suitable to represent the MLP-BP weights assuming that inputs and outputs are normalized between [-1,1]. For precision, precision values between [2^{-16}, 2^{-12}] were tested. Precision for fixed-point representation is constant while precision for floating point formats varies as discussed in section 4.2.2 and illustrated in Table 4.4 and Table 4.5 respectively. Note that fixed format fix1 - fix5 have comparable range/precision to floating-point formats float1 - float5. The same is true for fixed format fix11 - fix15 and float6 - float10. Fixed formats fix6 - fix10 have no comparable floating point format with similar range and precision but are added here to show the flexibility of the fixed-point formats.

We had two choices in implementing fixed-point multiplier: pipelined parallel and pipelined serial. Figure 4.18 shows that for different bit lengths these two implementations trade places as to which is the most efficient way to implement a fixed-point multiplier taking only space in consideration. Below 24 bit length, "parallel fixed-point multiplier" is more efficient, while above the 24-bit mark, "serial fixed-point multipliers" are more efficient. Since we are only concerned with < 20 bit length, our fixed-point multiplier will be implemented using pipelined parallel implementation.

Format #	Length (bits)	Integer (bits)	Fraction (bits)	Range	Precision
fix1	16	4	12	$[-8, 8-2^{-12}]$	2^{-12}
fix2	17	4	13	$[-8, 8-2^{-13}]$	2^{-13}
fix3	18	4	14	$[-8, 8-2^{-14}]$	2^{-14}
fix4	19	4	15	$[-8, 8-2^{-15}]$	2^{-15}
fix5	20	4	16	$[-8, 8-2^{-16}]$	2^{-16}
fix6	17	5	12	$[-16, 16-2^{-12}]$	2^{-12}
fix7	18	5	13	$[-16, 16-2^{-13}]$	2^{-13}
fix8	19	5	14	$[-16, 16-2^{-14}]$	2^{-14}
fix9	20	5	15	$[-16, 16-2^{-15}]$	2^{-15}
fix10	21	5	16	$[-16, 16-2^{-16}]$	2^{-16}
fix11	18	6	12	$[-32, 32-2^{-12}]$	2^{-12}
fix12	19	6	13	$[-32, 32-2^{-13}]$	2^{-13}
fix13	20	6	14	$[-32, 32-2^{-14}]$	2^{-14}
fix14	21	6	15	$[-32, 32-2^{-15}]$	2^{-15}
fix15	22	6	16	$[-32, 32-2^{-16}]$	2^{-16}

Table 4.4: Fixed-point Configurations

Format #	Length (bits)	Sign (bit)	Exponent (bits)	Mantissa (bits)	Range	Highest Precision
float1	15	1	2	12	$[2^{-10}-8, 8-2^{-10}]$	2^{-13}
float2	16	1	2	13	$[2^{-11}-8, 8-2^{-11}]$	2^{-14}
float3	17	1	2	14	$[2^{-12}-8, 8-2^{-12}]$	2^{-15}
float4	18	1	2	15	$[2^{-13}-8, 8-2^{-13}]$	2^{-16}
float5	19	1	2	16	$[2^{-14}-8, 8-2^{-14}]$	2^{-17}
float6	16	1	3	12	$[2^{-8}-32, 32-2^{-8}]$	2^{-15}
float7	17	1	3	13	$[2^{-9}-32, 32-2^{-9}]$	2^{-16}
float8	18	1	3	14	$[2^{-10}-32, 32-2^{-10}]$	2^{-17}
float9	19	1	3	15	$[2^{-11}-32, 32-2^{-11}]$	2^{-18}
float10	20	1	3	16	$[2^{-12}-32, 32-2^{-12}]$	2^{-19}

Table 4.5: Floating-point Configurations

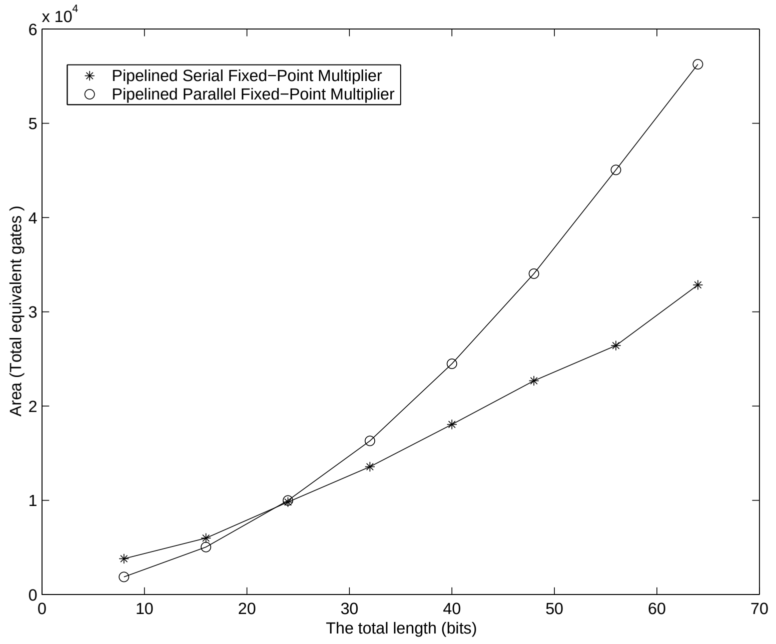

Figure 4.18: Fixed-point Multipliers with Different Lengths

4.5.3 Comparison of Various Formats: Area Requirements

Table 4.6 and Table 4.8 show the area requirements of all formats implemented as detailed in Table 4.4 and Table 4.5. The results show that a fixed-point adder is much more area efficient than floating-point adder regardless of the format used. This is illustrated in Figure 4.19 which shows the total area occupied by different adder configurations based on different fixed-point and floating-point formats. On the other hand, a floating-point multiplier is more area efficient than fixed-point multiplier regardless of the format used as illustrated in Figure 4.20.

For implementing MLP-BP, each processing element (PE) must implement multiplication and addition. It is clear that any successful implementation of MLP-BP on FPGAs must attempt to keep the computational parallelism observed at each

Format #	Gate Count			
	Type of Adder		Type of multiplier	
	Ripple Carry	Carry Look Ahead	Serial	Parallel
fix1	192	192	5,999	5,047
fix2	204	204	6,716	5,550
fix3	216	216	7,129	6,077
fix4	228	228	7,563	6,628
fix5	240	240	8,100	7,355
fix6	204	204	6,716	5,550
fix7	216	216	7,129	6,077
fix8	228	228	7,563	6,628
fix9	240	240	8,100	7,355
fix10	252	252	8,441	8,018
fix11	216	216	7,129	6,077
fix12	228	228	7,563	6,628
fix13	240	240	8,100	7,355
fix14	252	252	8,441	8,018
fix15	264	264	8,959	8,649

Table 4.6: Result from Fixed-point Operators

Format #	Gate Count	
	Pavle's adder	Pavle's multiplier
float1	1,837	2,694
float2	1,957	3,051
float3	2,065	3,469
float4	2,158	3,888
float5	2,257	4,372
float6	2,006	2,742
float7	2,117	3,099
float8	2,228	3,529
float9	2,342	3,936
float10	2,444	4,420

Table 4.7: Results from Pavle's Floating-point Operators

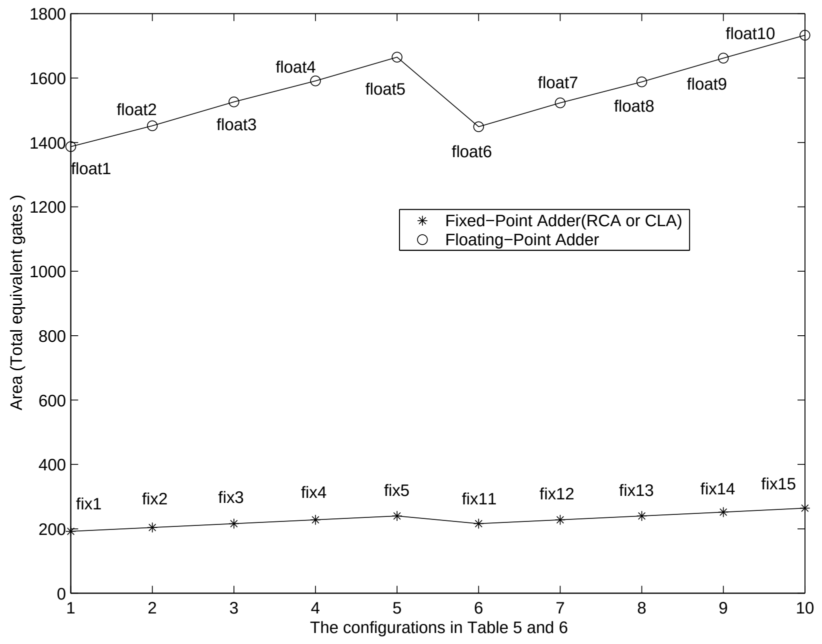

Figure 4.19: Fixed-point vs Floating-point Adder Configurations

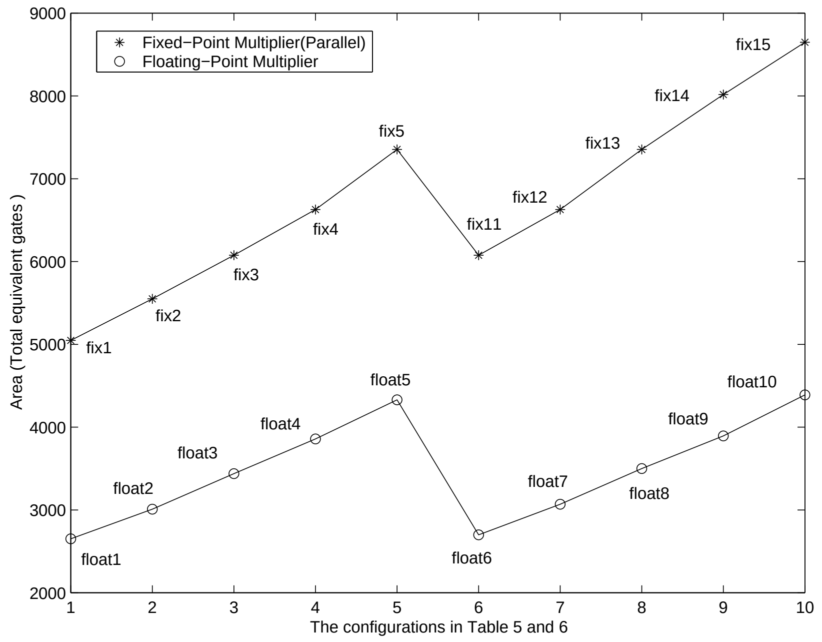

Figure 4.20: Fixed-point vs Floating-point Multipliers Configurations

layer by duplicating the necessary modules needed for each PE. As such it is desirable to compare various formats for implementing a single PE that combines one multiplier and one adder. Figure 4.21 and Figure 4.22 show that a PE based on floating-point formats are in general more area efficient than fixed-point formats. However, the target device plays an important role. In Figure 4.21 where the target device is Spartan-2E, the difference increases from 30% more gates for fix1 over float1 to 50% more gates for fix15 to float10. While in Figure 4.22, the fixed-point implementation is more efficient than floating-point implementation. These are quite interesting results and different than what has been reported in the literature [KNic02].

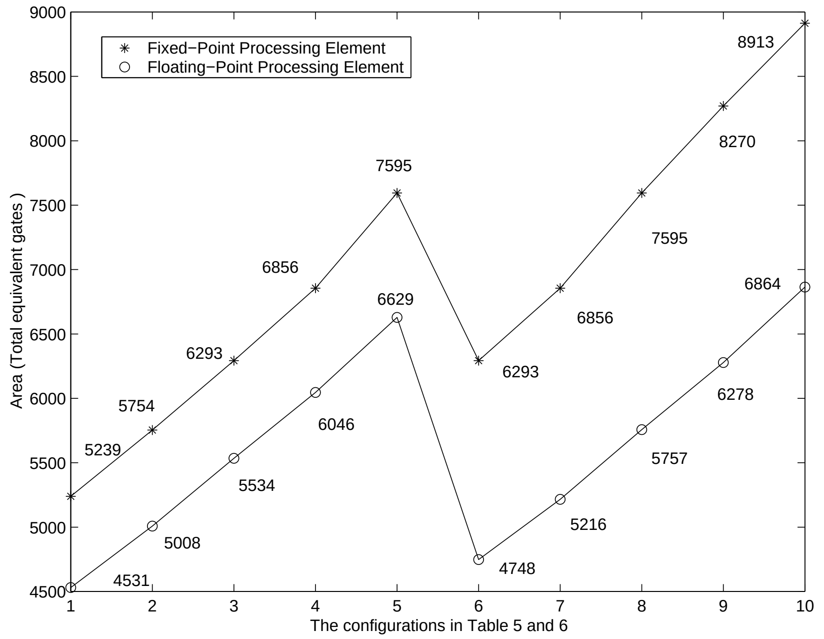

Figure 4.21: Fixed-point vs Floating-point Formats for a Single PE (Spartan-IIE)

The main reason for the difference between results of the Spartan-IIE and

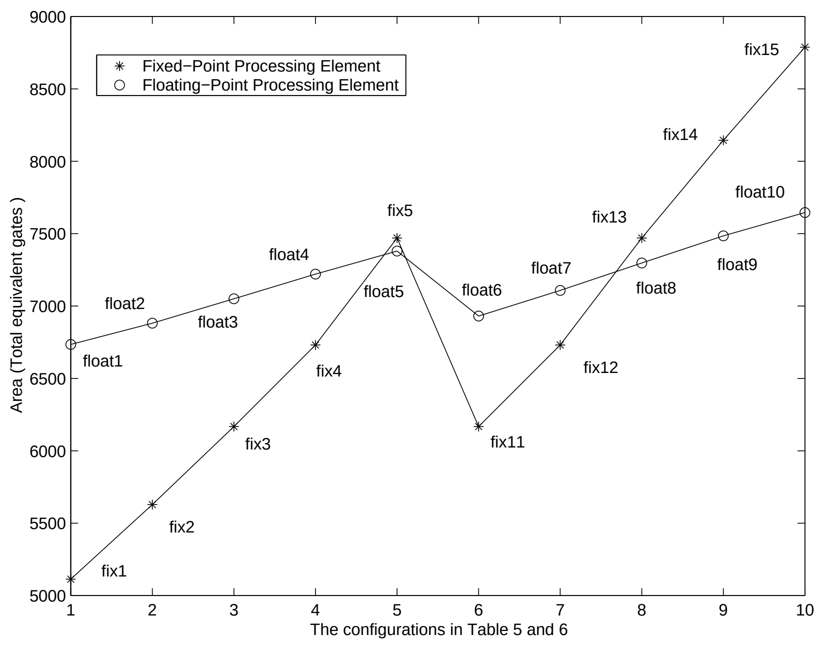

Figure 4.22: Fixed-point vs Floating-point Formats for a Single PE (Virtex-II)

Format #	Gate Count	
	Pavle's adder	Pavle's multiplier
float1	1,837	2,694
float2	1,957	3,051
float3	2,065	3,469
float4	2,158	3,888
float5	2,257	4,372
float6	2,006	2,742
float7	2,117	3,099
float8	2,228	3,529
float9	2,342	3,936
float10	2,444	4,420

Table 4.8: Results from Pavle's Floating-point Operators

Virtex-II FPAGs can be traced back to Figure 4.17 which shows that Spartan FPGA has more efficient implementation of floating-point multiplier than Virtex-II FPGA. Another observation from Figure 4.21 and Figure 4.22 is that there is almost no drawback of increasing the range between float5 and float10 while this is not the case for fixed-point formats. On the other hand, precision has a more direct impact on area usage since it has a strong impact on multiplier area usage.

While these results show that floating-point format offers a more area efficient implementation than fixed-point counterparts with similar precision and range, there are still a number of questions that remain to be investigated. As shown in Section 2.1.2 of Chapter 2, each ANN processing element includes a transfer function that could take the form of a sigmoid. A study that investigates the efficient format in implementing a sigmoid or any other transfer function needs to be carried out. It seems that a complete implementation of a MLP-BP can conclusively provide an answer as to which format should be used since fixed-point implementation has a significant advantage when more adders are needed.

Finally, this Chapter only considered the area efficiency vs precision and range requirements. Note that the comparison might not be 100% accurate due to different architecture between fixed-point and floating-point modules proposed. Another important consideration is speed which must also be investigated since it is quite reasonable to expect that future FPGAs will be big enough to make area used less of a factor than it is today.

4.6 A Pure Hardware XOR ANN

Some preliminary results were presented here for a pure hardware XOR ANN to evaluate the feasibility of implementing a large ANN onto a single Xilinx FPGA chip.

The MLP-BP network for the XOR problem has two input nodes, two hidden nodes, and one output node shown in Figure 4.23. Four different input pairs: (0,0), (0,1),(1,0), and (1,1) were applied to the network with corresponding target outputs: 0, 1, 1, and 0.

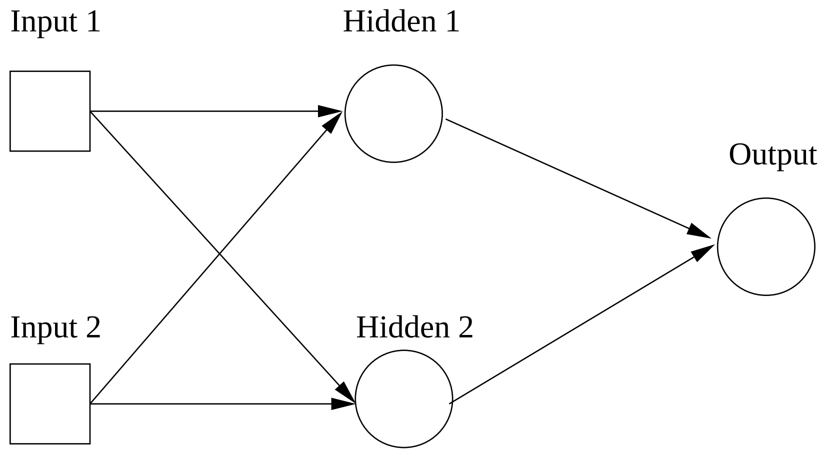

Figure 4.23: The MLP-BP Network for XOR problem

The hardware implementation shown in Figure 4.24 includes three parallel computation modules: "feedforward calculation", "backward calculation", , "updating" and a controller. Each neuron in the feedforward module shown in Figure 4.25 contains two fixed-point multiplier (parallel unsigned fixed-point multiplier inside) and one fixed-point adder and a 3-piecewise linear sigmoid function module.

The computational work of equations 2.10, 2.22 are implemented by parallel sub-

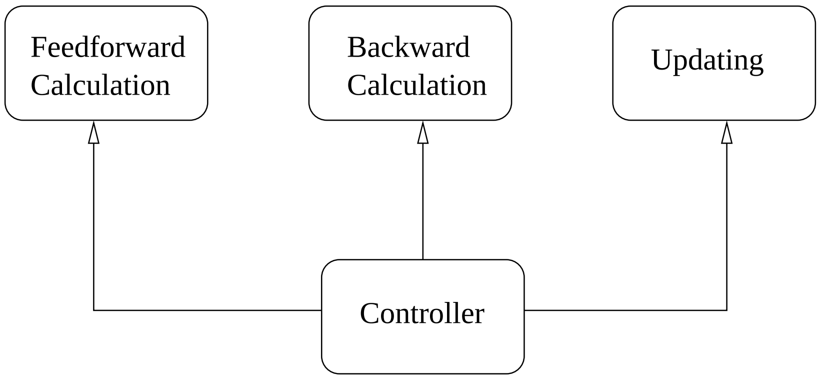

Figure 4.24: The Top View of the XOR ANN Implementation

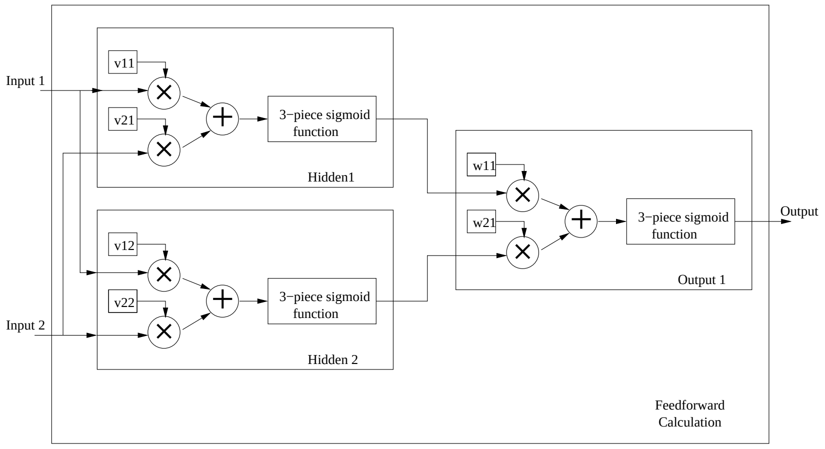

Figure 4.25: The Architecture inside of the Feedforward Module

modules inside of the "backward calculation" of Figure 4.24. The computational work of equations 2.6, 2.9, 2.17 and 2.21 are implemented by parallel sub-modules in side of the "updating" of Figure 4.24. In other words, the mathematical calculations of those equations are implemented in parallel at the same time.

Results are shown in Table 4.9 where three different fixed-point formats are tested. The FPGA area consumed is presented by how the amount of slices used. Notice, the Xilinx Virtex-II FPGA (used to implement the MLP-BP network for solving the face recognition in the next Chapter) only contains 10,752 slices. It is obvious that the pure hardware implementation of a XOR ANN is fairly huge and infeasible to be implemented onto a Virtex-II XC2V2000 device. In order to implement a large network, a hardware/software co-design approach is followed in the next Chapter.

Fixed-point Formats (sign-integer-fraction)	FPGA Area (slices)
1-3-16	9,039
1-4-16	9,838
1-5-16	10,525

Table 4.9: The Results of XOR ANN

4.7 Summary

In this Chapter the affect of arithmetic representation formats on implementing ANN on FPGAs was investigated. The main focus was to examine the trade offs between precision and range of various formats that can target FPGA resources. The basic ANN processing elements include multiplication and addition operations.

As such, twenty five floating and fixed-point formats were tested where a multiplier and an adder were implemented on an FPGA and their area requirement were compared. The results show that for implementing a Multilayer perceptron neural network, floating-point formats offer more area efficient implementation without penalty on precision or range. The results also show that the target FPGA device could have a major impact on the resources required. However, implementing a large ANN onto a single FPGA is still not feasible. In order to balance performance and flexibility, a Hardware/Software Co-design approach should be followed. The next Chapter introduces a flexible methodology that takes advantage of a soft processor core and dedicated hardware to efficiently implement a MLP-BP ANN onto a Xilinx Virtex-II FPGA.

Chapter 5

A Hardware/Software Co-design Approach

To evaluate the feasibility of implementing a large MLP-BP ANN on an FPGA, the face recognition problem was chosen as a benchmark. The network constructed for this application contains 400 input nodes, 8 hidden nodes, and 4 output nodes. Given the information in the previous Chapter, a pure hardware implementation of such a network is not feasible on a Xilinx Virtex-II FPGA (2 million system gates). Instead, a hardware/software co-design approach is followed, where the most computationally intensive part (the updating of weights and thresholds) of the MLP-BP is implemented by hardware circuitry and the rest by software running on a processor. Consequently, the flexibility of the processor and the performance of the hardware could be balanced. Industry's fastest soft processor IP core, *MicroBlaze* (Appendix C), and dedicated hardware modules were implemented by following the Xilinx embedded system design flow (Appendix B) on a Virtex-II XC2V2000 FPGA of a Xilinx Multimedia board (Appendix D). Results obtained indicate that performance in terms of speed was improved by 1.69 (Amdahl's law) with the flexibility preserved to some extent.

5.1 Target application description

5.1.1 Working Strategy

The two phases of this face recognition system are the training phase and the testing phase. In the training phase, a three-layer perceptron with a *backpropagation* algorithm was used as a supervised learning algorithm as shown in Figure 5.1. *Image Sender* is used to fetch images from the training image list where images are used for training ANN (Figure 5.1). The output of *Image Sender* is then directed to the following subsystems: *Target Generator* and *Learning System*. The *Target Generator* encodes a certain image to its corresponding target output that the *Learning System* requires. The output-encoding strategy of this system is described in greater detail later. The *Target Generator* is similar to a teacher who tells the *Learning System* what should be outputted for different face images in the training image list. This work is done by calculating the difference between the actual output of the *Learning System* and target output of the *Target Generator*. This difference is then fed back to the *Learning System* to decrease the error.

Following the training phase, the testing phase is initiated (Figure 5.2). where the *Image Sender* sends images to the trained learning system. The output of the trained *Learning System* is fed into the *Output Interpreter*, where the output of the *Learning System* is translated into an identification of faces. In this case,

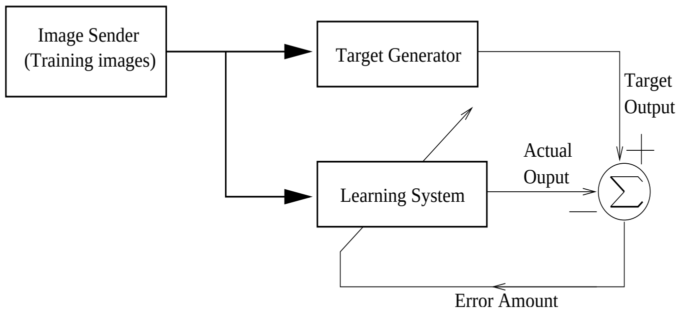

Figure 5.1: Supervised Training Diagram

the system is used to recognize different persons. In other cases, the system can recognize different expressions, head directions, e.t.c. Finally the interpreted results from the *Output Interpreter* are displayed to the user.

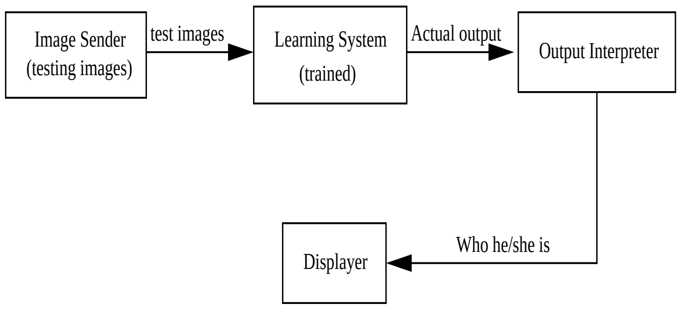

Figure 5.2: Testing Process Diagram

5.1.2 Face Images

The face images were collected from Professor Tom M. Mitchell's Machine Learning course website [httpd] of the School of Computer Science, Carnegie Mellon University. This face image database contains images of 20 people, each individual with 32 images varying in expression (happy, sad, angry, neutral), the direction in which they are looking (left, right, straight ahead, up), and whether or not their eyes are open or closed. In total, 624 grayscale images in PGM format [httpc] were collected, each with a resolution of 120*128 pixels, with each image pixel described by a grayscale intensity value between 0 (black) and 255 (white). Some of the selected face images are shown in Figure 5.3. The first four images starting at the left-hand side of the the first row represent persons with different expressions, and the next 4 images in the first row represent persons with different directions. The persons in the second row show expressions and directions corresponding with those in the first row, with their eyes closed.

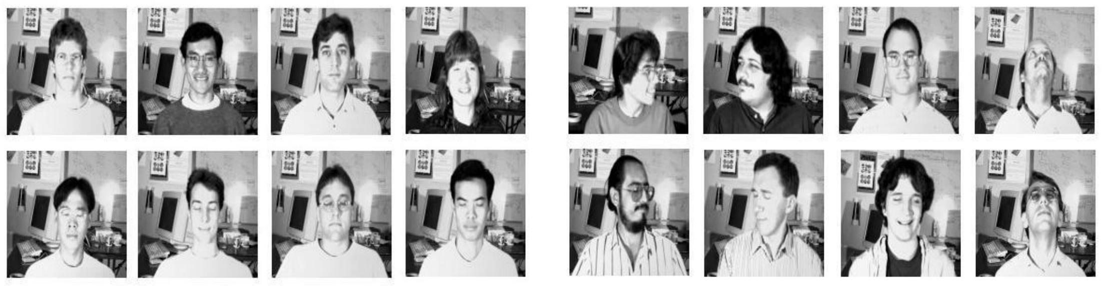

Figure 5.3: An Example of Face Images

5.1.3 Input Encoding

As illustrated in Figure 5.4, the face images are fed into the MLP-BP network. However, the kind of representation of the face images that should be used raises an issue. In general, a face image could be preprocessed so that some features are extracted. Either features (e.g., curves, edges, etc.) or the raw intensity value of the image could be used as an appropriate representation of the image fed to the

ANN. The latter was chosen since Professor Tom M. Mitchell highlighted the following: *"the difficulty of using features is that it would lead to a variable number of features per image, whereas the ANN has a fixed number of input units [Mitc97]"*. The intensity values of each pixel of a certain face image is from 0 to 255. The *backpropagation* algorithm introduced in Chapter 2 was used on this three-layer perceptron ANN to identify different persons. In addition to the image representation problem, another issue identified is the image size to be used. The original image size is an accurate representation of the face images but can be excessive and a burden on the processing time. The image size used in this research was 20*20, which is the coarse resolution summary of the original image. As a result, the number of input nodes and the weights between input layer and hidden layer of the ANN were reduced drastically. Meanwhile, the sufficient classification performance of the images was preserved. In Prof. Tom. M. Mitchell's C language implementation, the image size was reduced to 32*30 which proved to be sufficient. In this research, due to limitations in the size of the Block RAM, the image size was reduced to 20*20. In order to verify the validation of this image size, a comparison was done on 32*30 and 20*20 separately. In each case, the ANN was trained with 320 images of 20 different persons with 4 directions (left, right, straight ahead, up) and two expressions (happy and sad) with eyes open and closed. When the testing images were within the training images, the accuracy rate reached 94.8% for both. Also for the generalization performance (which means the testing images were not from the training images) both 32*30 and 20*20 image sizes reached around 90% accuracy rate (32*30 image had 3% better than 20*20 image). From this experimental result, 20*20 image size was considered acceptable for the face recognition problem. The work of scaling images from 120*128 to 20*20 was done by Gimp1.3 [httpa] tool under SUSE9.0 linux operating system.

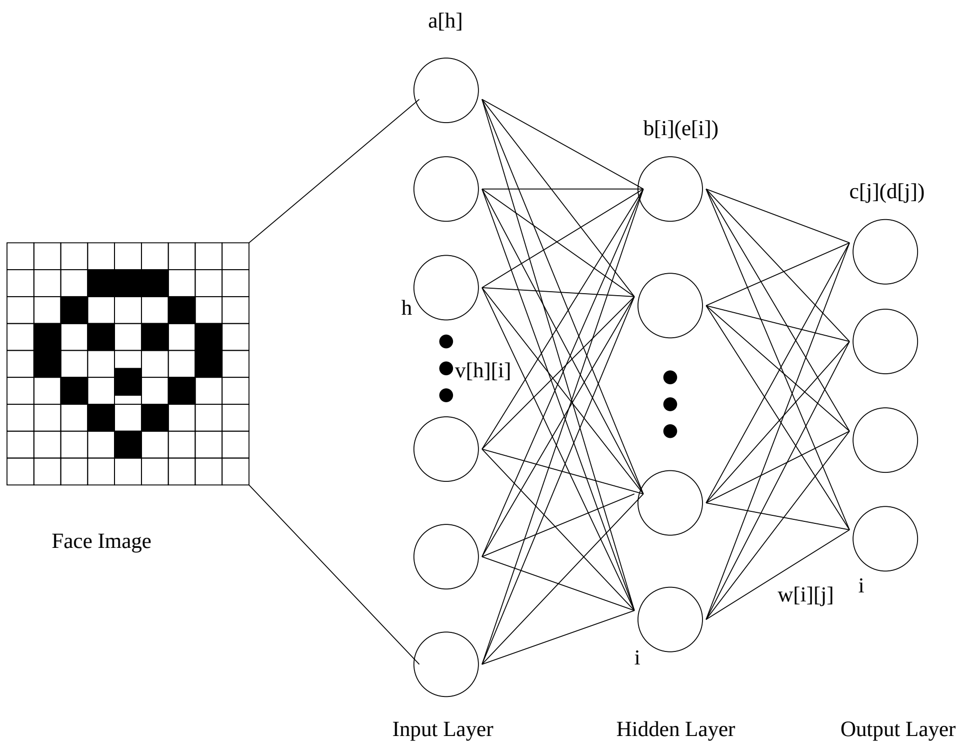

Figure 5.4: Mapping an Image to ANN

5.1.4 Output Encoding

The system constructed has the ability to recognize four different persons at a time. To achieve this, different output encoding strategies could be used. If one output is used, then differentiating 4 persons can be achieved by 4 different target values: 0.2, 0.4, 0.6 and 0.8. On the other hand, if 4 output nodes are used, the target outputs could be in the form of the following vectors: (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0) and (0, 0, 0, 1). Other output encoding strategies also exist. In this thesis, an encoding strategy with 4 output nodes was used due to the following: Firstly, a four output encoding strategy gives the ANN more degrees of freedom to perform the classification than one output encoding strategy. Secondly, as mentioned in Mitchell's book – *Machine Learning* [Mitc97], *"the difference between the highest-valued output and the second-highest can be used as a measure of the confidence in the network prediction (ambiguous classifications may result in near or exact ties)"*. Furthermore, 0 and 1 in the vector are substituted by 0.1 and 0.9 separately. For example, (0.9, 0.1, 0.1, 0.1) is used as the target vector for the first person instead of (1, 0, 0, 0). This modification is because the active function (logistic sigmoid function) used for the ANN cannot reach an exact value of 0 or 1.

5.1.5 Network Graph Structure

The ANN constructed in this project is a three-layer perceptron network as shown in Figure 5.4. Two layers of sigmoid units (one hidden layer and one output layer) are used in this network. One hidden layer is sufficient in most applications, and more hidden layers are generally not recommended due to the long training time involved [Mitc97]. There are 400 nodes in the input layer which are waiting to receive the grayscale intensity values of 20*20 face images. The number of 8 hidden nodes was chosen based on experimentation.

5.2 A Pure Software Implementation

The pseudo code for the training process and the testing process is shown in Figure 5.5. Firstly, the *opb_timer* is an instance of **OPB Timer/Counter** [httph], which is a module used to calculate the time consumed by the program running on *MicroBlaze*. Secondly, the neuron parameters (weights and thresholds) are initialized to some random value. The *epoch_num* is a variable that determines the stopping criteria. Each image is trained at each iteration. During the training phase, three functions are executed. The *forward()* phase is used to calculate the node value in the hidden layer and the output layer; the *backward()* phase is used to calculate the error associated with the nodes in the output layer and the hidden layer, and the *update()* phase is used to update the weights and the thresholds. Following the training process, the time consumed during this process is read from *opb_timer*. The testing process is then initiated. In this phase, a certain image is selected from the training image list and fed into the ANN. The value of the output nodes are finally interpreted to identify the person.

5.2.1 Initialization of the *opb_timer*

The *opb_timer* was set to the "*generate-mode*" and started from zero. There are three steps in starting the *opb_timer*: initializing the load register with zero, setting the mode to *generate mode* (also loading zero to the counter by TCSR) and starting the *opb_timer*. To stop the *opb_timer*, the ENT bit of TCSR is disabled. The process of starting and stopping this timer is shown in Figure 5.6 and Figure 5.7 respectively.

```
Start_timer();
Randomization();
//Training process
while(i<epoch_num)
{
  for(j=0;j<number of training images;j++)
  {
    Initial input nodes to the values of image j;
    Set target value for image j;
    forward();//implemented by C
    backward();//implemented by C
    update();//implemented by C
  }
}
Stop_timer();
//Testing process
Initial input nodes to the value of a certain testing image;
forward();
print and interpret result;
```

Figure 5.5: Pseudo Code of Backpropagation Algorithm for Face Recognition

```
void Start_timer()
{
  //initialize Load Register TLR0 with 0
  XIo_Out32(XPAR_OPB_TIMER_TLR0,0x00000000);
  //load 0 into the counter0 and configure it count up
  XIo_Out32(XPAR_OPB_TIMER_TCSR0,0x00000020);
  //start up the counter0
  XIo_Out32(XPAR_OPB_TIMER_TCSR0,0x00000080);
}
```

Figure 5.6: Pseudo Code for Starting opb_timer

```
void Stop_timer()
{
  //read out the timer
  cycles=XIo_In32(XPAR_OPB_TIMER_TCR0);
  //stop the timer
  XIo_Out32(XPAR_OPB_TIMER_TCSR0,0x00000000);
}
```

Figure 5.7: Pseudo Code for Stopping opb_timer

5.2.2 Randomization

Preceding the training process, several parameters have to be randomized. including: (i) all the weights between the input layer and the hidden layer; (ii) all the weights between the hidden layer and the output layer; (iii) all the thresholds associated with the hidden nodes, and the thresholds associated with the output nodes. The pseudo code is shown in Figure 5.8. NODE_NUM_I represents the number of input nodes, NODE_NUM_H the number of hidden nodes, and NODE_NUM_O the number of output nodes. The value v[i][h] is the weight between the input node i and the hidden node h, w[h][o] is the weight between the hidden node h and the output node o, threh[h] is the threshold value of the hidden node h, and threo[o] is the threshold value of the output node o.

The function *rand()* is an ISO C Random Number Function which returns a value in the range from 0 to 2147483647.

5.2.3 The forward() function

Following the neuron parameters initialization, the feedforward phase proceeds. In this phase, the value of the hidden and the output node is calculated.

```
void Randomization()
{
  for(i=0;i<NODE_NUM_I;i++)
  {
    for(h=0;h<NODE_NUM_H;h++)
    {
      if(rand()%2==1)
        v[i][h]=rand() mod 5000/10000.0;
      else
        v[i][h]=-rand() mod 5000/10000.0;
    }
  }
  for(h=0;h<NODE_NUM_H;h++)
  {
    for(o=0;o<NODE_NUM_O;o++)
    {
      if(rand()%2==1)
        w[h][o]=rand() mod 5000/10000.0;
      else
        w[h][o]=-rand() mod 5000/10000.0;
    }
  }
  for(h=0;h<NODE_NUM_H;h++)
  {
    threh[h]=rand() mod 5000/10000.0;
  }
  for(o=0;o<NODE_NUM_O;o++)
  {
    threo[o]=rand() mod 5000/10000.0;
  }
}
```

Figure 5.8: Pseudo Code for Random Parameters Process

A sigmoid function was used in order to solve non-linear problems. Two popular sigmoid functions, *Logistic* and *Hyperbolic tangent*, were used and are described in Section 2.1.2 of Chapter 2. The *forward()* function is shown in Figure 5.9 based on the Logistic function. If the Hyperbolic tangent function is used, the line above each of the commented lines is replaced by a different commented line. In both cases, a[i] is the value of the input node i, b[h] is the value of the hidden node h, and c[o] is the value of the output node o.

The equations for the Logistic and Hyperbolic functions are as follows:

$$f(x) = \begin{cases} \frac{1}{1+e^{-x}} & \text{Logistic Function} \\ \frac{1-e^{-x}}{1+e^{-x}} & \text{Hyperbolic Tangent Function} \end{cases}$$

5.2.4 The backward() function

As explained earlier in Chapter 2, the error value d_o is determined by:

$$d_o = \begin{cases} \beta(1-c_o)c_o(c_o^k - c_o) & \text{if Logistic Function is used} \\ \beta(1-c_o^2)(c_o^k - c_o)/2 & \text{if Hyperbolic Tangent Function is used} \end{cases}$$

The pseudo code for the *backward()* function is shown in Figure 5.10. The value d[o] is the error value of the output node i, and e[h] is the error value of the hidden node h.

```
void forward()
{
  for(h=0;h<NODE_NUM_H;h++)
  {
    temp=0.0;
    for(i=0;i<NODE_NUM_I;i++)
    {
      temp=temp+a[i]*v[i][h];
    }
    b[i]=1.0/(1.0+exp(-β*(threh[h]+temp)));
    // b[i]=(1.0-exp(-β*(threh[h]+temp)))/(1.0
     +exp(-β*(threh[h]+temp)));
  }
  for(o=0;o<NODE_NUM_O;o++)
  {
    temp=0.0;
    for(h=0;h<NODE_NUM_H;h++)
    {
      temp=temp+b[h]*w[h][o];
    }
    c[j]=1.0/(1.0+exp(-β*(threo[o]+temp)));
    //c[i]=(1.0-exp(-β*(threo[o]+temp)))/(1.0
     +exp(-β*(threo[o]+temp)));
  }
}
```

Figure 5.9: Pseudo Code for forward() Function

```
void backward()
{
  for(o=0;o<NODE_NUM_O;o++)
  {
    d[o]=$\beta$(1-c[o])c[o]($c^k$[o]-c[o]);
    //d[o]=$\beta$(1-c[o]*c[o])($c^k$[o]-c[o])/2.0;
  }
  for(h=0;h<NODE_NUM_H;h++)
  {
    temp=0.0;
    for(o=0;o<NODE_NUM_O;o++)
    {
    temp=temp+$\beta$(1-b[h])b[h]w[h][o]d[o];
    //temp=temp+$\beta$(1-b[h]*b[h])w[h][o]d[o]/2.0;
    }
    e[h]=temp;
  }
 }
```

Figure 5.10: Pseudo Code for backward() Function

5.2.5 The update() function

The connection weights between the hidden layer node 'h' and the output layer node 'o' are updated based on the calculated value of the hidden node b[h] and the error value of the output node d[o]. On the other hand, connection weights between the input layer node 'i' and the hidden layer node 'h' are updated based on the calculated value of the input node a[i] and the hidden error value e[h]. In order to control the updating speed, a learning rate α is introduced. The updating for the threshold values of the hidden nodes and the output nodes depends on the error value of the corresponding node.

The pseudo code for software implementation of the *update()* function is shown in Figure 5.11.

5.2.6 ANN Parameters

The ANN parameters (Table 5.1) include: (i) the initial range of the weights and thresholds; (ii) the learning rate value; (iii) the epoch numbers; (iv) the input node number; (v) the hidden node number and (vi) the output node number. These parameters were determined from 100 experiments conducted and proved to be optimal values.

5.2.7 System Architecture

The reconfigurable system constructed is based on a MicroBlaze development system shown in Figure C.1. The system mainly include a *MicroBlaze* soft processor and miscellaneous processor peripherals. The *MicroBlaze* soft processor can be

```
void update()
{
  for(o=0;o<NODE_NUM_O;o++)
  {
    for(h=0;h<NODE_NUM_H;h++)
    {
      w[h][o]=w[h][o]+$\alpha$b[h]d[o];
    }
    threo[o]=threo[o]+$\alpha$d[o];
  }
  for(h=0;h<NODE_NUM_H;h++)
  {
    for(i=0;i<NODE_NUM_I;i++)
    {
      v[i][h]=v[i][h]+$\alpha$a[i]e[h];
    }
    threh[h]=threh[h]+$\alpha$e[h];
  }
}
```

Figure 5.11: Pseudo Code for Software update() Function

Parameters	Description
weights	randomized between -0.5 and 0.5
thresholds	randomized between 0 and 0.5
learning rate	0.01
epoch number	2000
input nodes	400
hidden nodes	8
output nodes	4

Table 5.1: Chosen Parameters of ANN

realized by 900 logic cells of a Xilinx Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Spartan-II, or Spartan-IIE devices. The area consumed by processor peripherals varies by different applications.

The pure *MicroBlaze* system used to solve the face recognition problem is shown in Figure 5.12. There are seven IP cores inside this system, namely *MicroBlaze core*, *OPB Block RAM controller*, *OPB Block RAM*, *OPB JTAG_UART*, *OPB Timer/Counter*, *OPB UART Lite* and *OPB General Purpose Input/Output (GPIO)*. The detail information of each IP core is listed in Appendix F.

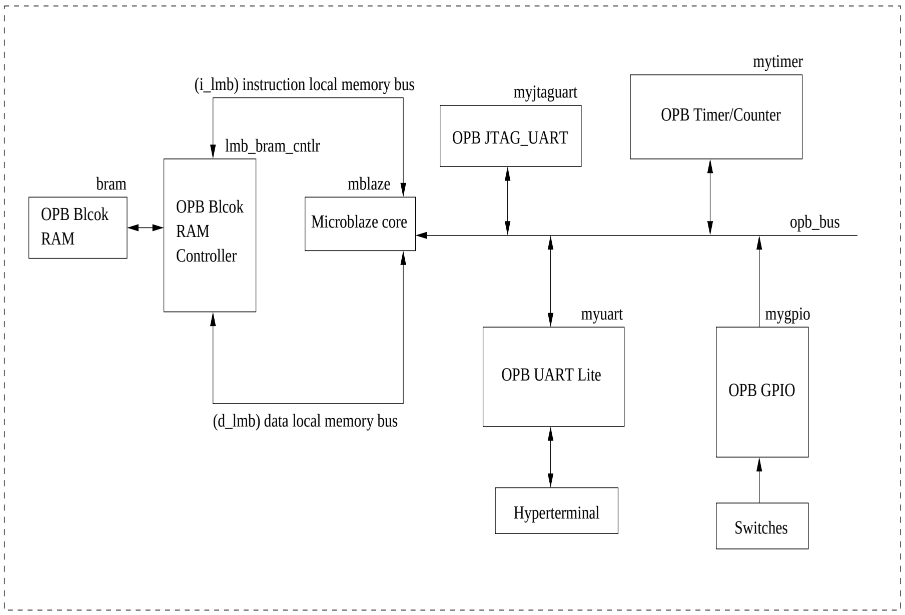

Figure 5.12: *MicroBlaze* System for Face Recognition

The *backpropagation* algorithm is coded in C programming and stored in Block RAM to solve the face recognition problem. The face images (currently 4 images are stored) are stored in Block RAM to avoid a relatively long time to access external memory. OPB JTAG_UART is used to debug the system's functionality. OPB

Timer/Counter is used to calculate the time duration consumed by each module. The OPB UART Lite is used to communicate between the PC and the embedded system on FPGA. OPB GPIO is used as control inputs from the outside world.

5.2.8 Results

The results include two aspects: FPGA usage and profiling. In FPGA usage, the FPGA resource consumed by each IP core is presented. In profiling, the time consumed by each function is presented. There are two profiling results. The first (Table 5.3) is the profiling of Prof. Tom's face recognition code [httpe] running on Intel Pentium4 2.4 GHz and 1 GB RAM PC with Linux redhat9.0 operating system. The second (Table 5.4) is the profiling of the *MicroBlaze* system that runs the same C-code.

- FPGA usage: The FPGA usage of each IP core of Figure 5.12 is shown in Table 5.2. The result was collected from Xilinx floorplanner tool. "MULTs" is Xilinx Virtex-II dedicated 18×18-bit multiplier. "Function Generator" is the 4-input LUT and "FlipFlops" is the storage element. "32-bit Dual RAM" is the internal register of *MicroBlaze* and "16-bit Mem." is the 16-bit register.

- Profiling: The profiling result for Prof. Tom's C code is presented in Table 5.3 (70 images each with 32*30 resolution trained by 100 epochs). The profiling result for C code running on *MicroBlaze* is presented in Table 5.4 (4 images, each one with 20*20 resolution trained by 1 epoch).

  From the results shown in Table 5.3, *bpnn_adjust_weights* function could be a candidate to be implemented onto hardware. However, results in Table 5.4

	Block RAM	MULTs	16-bit Mem.	32-bit Dual RAM	Fun Gen.	Carry Sym.	Flip Flops
System	56	3	101	64	1558	187	1297
myuart	0	0	19	0	76	25	62
mytimer	0	0	0	0	283	0	313
mblaze	0	3	73	64	931	143	829
opb_bus	0	0	1	0	167	2	11
lmb_bram_cntlr	0	0	0	0	5	0	1
myjtaguart	0	0	18	0	86	14	79
bram	56	0	0	0	0	0	0

Table 5.2: FPGA Usage

% time	Cumulative seconds	Self seconds	Name
71.00	4.04	4.04	bpnn_adjust_weights (equation 2.6, 2.9, 2.17, 2.21)
26.19	5.53	1.49	bpnn_layerforward (equation 2.1, 2.2)
1.76	5.63	0.10	load_input_with_image
0.35	5.65	0.02	squash
0.18	5.66	0.01	bpnn_hidden_error (equation 2.22)
0.18	5.67	0.01	bpnn_save
0.18	5.68	0.01	img_getpixel
0.18	5.69	0.01	load_target
0.00	5.69	0.00	backprop_face

Table 5.3: The Profiling Result for Prof. Tom's Code

% time	Cycles	Seconds	Name
54.10	2077492	0.0207	forward (equation 2.1, 2.2)
0.81	31283	0.0003	backward (equation 2.10, 2.22)
45.07	1730347	0.0173	update (equation 2.6, 2.9, 2.17, 2.21)

Table 5.4: The Profiling result C Code Running on *MicroBlaze*

indicate that the feedforward calculations and updating weights consume a similar computation time. The final decision was made in which the process of updating weights and thresholds is implemented onto dedicated hardware and the remaining functions were realized by *MicroBlaze*. The following Section shows the implementation of a system containing a *MicroBlaze* and dedicated hardware modules.

5.3 Hardware Update Module (HUM)

The pure soft core implementation of ANN for solving the face recognition problem was presented in Section 5.2. It is important to understand that the system functionality can be easily modified by changing the C code running on the soft core *MicroBlaze*. However, speed is an important system specification that can be achieved using dedicated hardware modules. In order to balance speed and flexibility, both hardware and software were implemented to create the face recognition system. In the current implementation, the *update()* function was implemented by pure hardware while the remaining functions were implemented in software. This decision was made based on the profiling result of Prof. Tom M. Mitchell's C implementation on a PC station where the *"adjusting weights"* process consumed the most computation time as shown in Table 5.3.

5.3.1 System Architecture

This Section introduces an embedded computing system where both a *MicroBlaze* processor and a dedicated hardware module named *HUM* are integrated as shown in Figure 5.13. The communication between *HUM* and *MicroBlaze* is through two Fast Simplex Link (FSL) channels. A general description of *HUM* and a detailed description of FSL channel will be presented next.

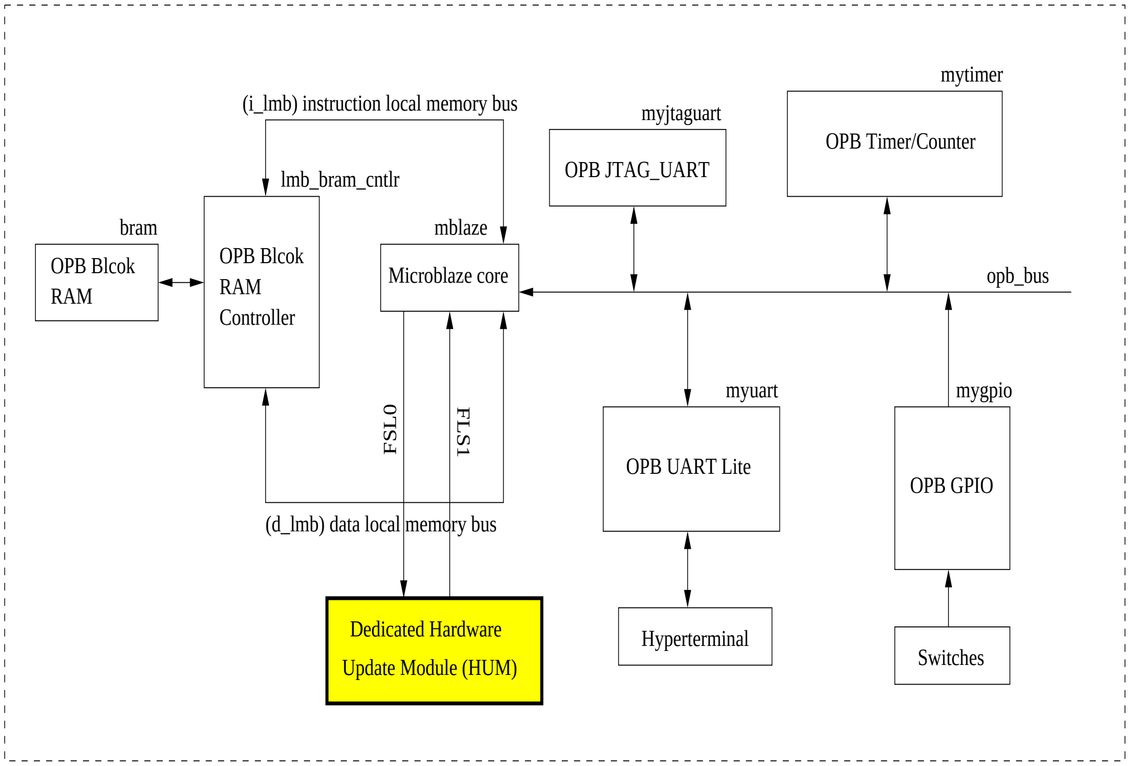

Figure 5.13: System Architecture

5.3.1.1 Hardware Update Module (*HUM*)

The block diagram of *HUM* is shown in Figure 5.14. The 4 *UPDATE UNITs* are designed to update 4 weights or thresholds simultaneously. The module *"counter1"* is used to control the storage process of 16 parameters (for updating 4 weights or 4 thresholds). *Finite State Machine* is used to control the *UPDATE UNITs* and *counter2*. The module *"counter2"* is used to control the sending process of four updated weights (thresholds). The left-hand side of the block diagram is connected with *MicroBlaze* through an FSL0 channel where *MicroBlaze* is master and *HUM* is slave. The parameters used for updating weights (thresholds) are sent from *MicroBlaze* to *HUM*. The right-hand side of the block diagram is connected to *MicroBlaze* through FSL1, where the *HUM* is a master and the *MicroBlaze* is slave. The updated weights (thresholds) are sent back from *HUM* to *MicroBlaze*. Detailed information on *FSL channel*, *FSL0/FSL1 interface of HUM* is presented in the following sub-sections.

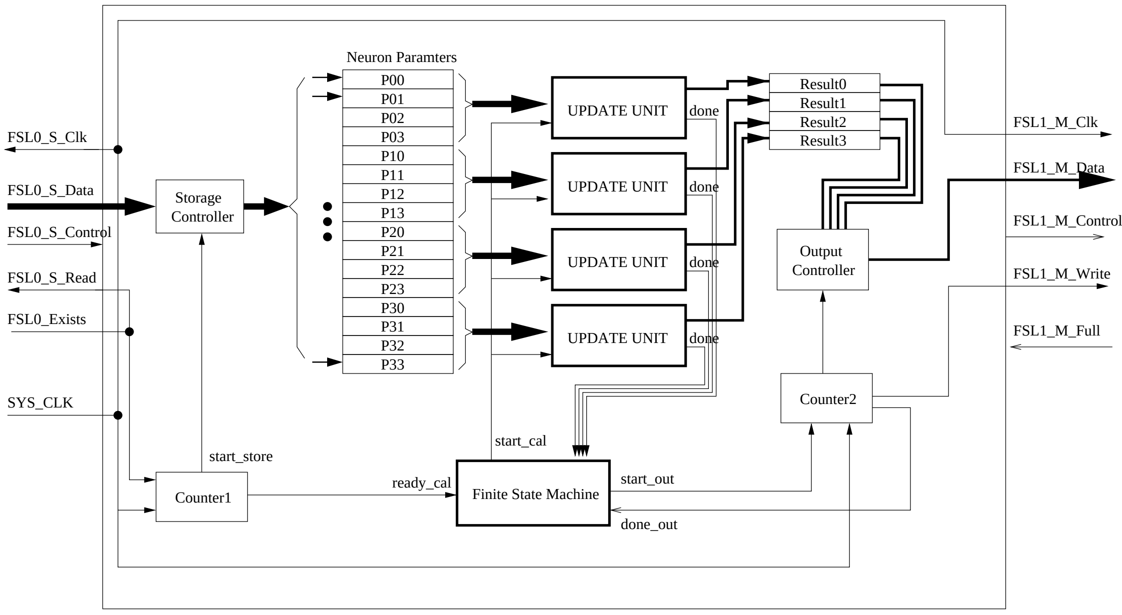

Figure 5.14: The Block Diagram of *HUM*

5.3.1.2 Fast Simplex Link (FSL) channel

The *MicroBlaze* software contains eight input and output FSL interfaces which are dedicated unidirectional point-to-point data streaming interfaces. The FSL interfaces on *MicroBlaze* are 32-bit wide. Further, the same FSL channels can be used to transmit or receive either control or data words. A separate bit indicates whether the transmitted (received) word is control or data information. The performance of the FSL interface can reach up to 300MB/sec. The FSL bus is driven by one Master and drives one Slave. Figure 5.15 shows the mechanisms of the FSL bus system and the available signals.

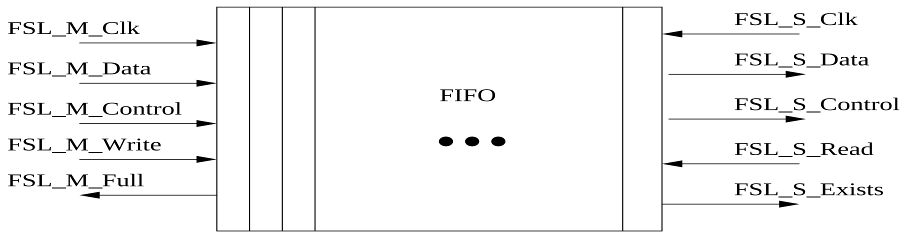

Figure 5.15: FSL Interface

FSL peripherals may be created as a Master or a Slave to the FSL bus. A master peripheral connected to the left-hand side of Figure 5.15 transmits data and control signals onto the FSL. All slave peripherals connected to the right-hand side of Figure 5.15 receive the data sent from the master peripheral. In this architecture, the *HUM* was connected with the *MicroBlaze* processor through two FSL channels: FSL0 and FSL1, shown in Figure 5.16.

The two FSL predefined C-functions below are used to transfer information between FSL master and FSL slave, where *val* is a 32-bit binary number, and *id* identi-

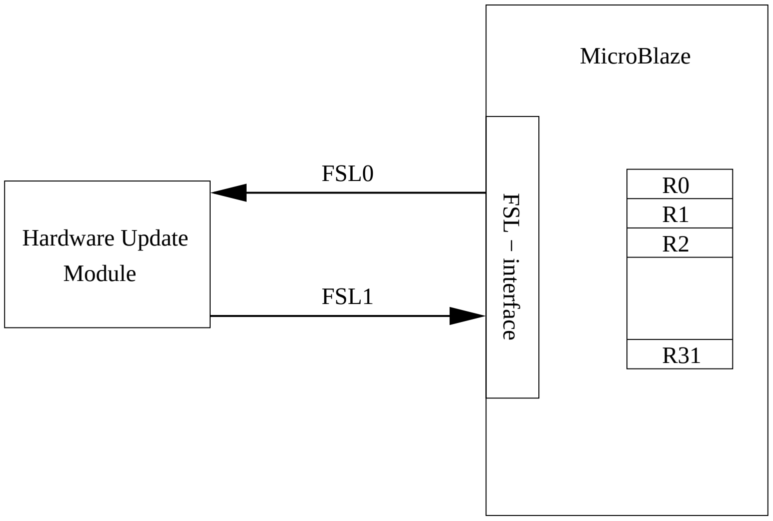

Figure 5.16: Connecting *HUM* via FSL Interface onto *MicroBlaze*

fies the channel. For example, *MicroBlaze* uses $microblaze\_nbwrite\_datafsl(0x00000000, 0)$ to send decimal 0 to *HUM* through FSL0 and $microblaze\_nbwrite\_datafsl(reg, 1)$ to read the value from *HUM* and store it to a register "reg".

//Non-blocking Data Read and Write to Local Link no. id

$microblaze\_nbread\_datafsl(val, id);$

$microblaze\_nbwrite\_datafsl(val, id);$

5.3.1.3 FSL0/FSL1 interface to HUM

The left-hand side of Figure 5.14 is the interface to FSL0 and the right-hand side is the interface to FSL1. There are several input/output ports included in the interfaces of both FSL0 and FSL1 on the *HUM* module. The point-to-point protocol of FSL0 and FSL1 is presented in detail as follows:

1. **FSL0 interface on HUM:** The parameters sent from the *MicroBlaze* pro-

cessor are stored in the FIFO buffer of FSL0. Whenever data is available in the FIFO buffer, *FSL0_S_Exists* signal changes from '0' to '1'. *FSL0_S_Read* is an output signal from *HUM* to FSL0 where '1' means *HUM* has read the first data in the FIFO buffer. This signal is used by FSL0 to discard the most first data in FIFO. If *FSL0_S_Read* is '0', the data is stored in FSL0 FIFO buffer.

2. **FSL1 interface on HUM:** As mentioned earlier, the *HUM* module has the ability to update 4 weights (thresholds) simultaneously. When 4 updated weights (thresholds) are ready to be sent back to the *MicroBlaze* processor, *FSL1_M_Write* becomes '1' for four clock cycles, and each one of the four updated weights (thresholds) is sent out on *FSL1_M_Data* port, one per cycle. If FSL1 FIFO buffer is full, *FSL1_M_Full* changes from '0' to '1'.

5.3.2 System Functionality

In this section, the working mechanism of the embedded computing system is described by three stages. In the first stage, parameters are sent from the *MicroBlaze* processor to FSL0 channel, and then *HUM* stores the parameters to its local registers. In the second stage, parameters received from *MicroBlaze* are sent to four *UPDATE UNITs* where new weights (thresholds) are calculated. This is done by one floating-point adder and two multipliers inside *UPDATE UNIT*. The updated results are stored in another four local registers. In the third stage, updated results are sent from *HUM* to FSL1 and consequently the *MicroBlaze* stores the results. This process continues until all weights and thresholds are updated.

5.3.2.1 Stage #1

There are two sub-stages that need to be explained. Firstly, the *MicroBlaze* processor sends parameters to FSL0 channel. Secondly, *HUM* loads the parameters and stores them in the local registers.

1. **From *MicroBlaze* to FSL0:** As mentioned earlier, the *MicroBlaze* uses two functions to write data to and read data from the FSL channel. When updating starts, the *MicroBlaze* processor needs to send parameters to *HUM* for weight and threshold updating. Four parameters are needed to update a certain weight. For example, to update $w[i][j]$, four parameters ($w[i][j]$, $b[i]$, $d[j]$, af) are needed. The parameter $w[i][j]$ is the old value of weight between hidden node i and output node j; $b[i]$ is the activation value of hidden node i; $d[j]$ is the error value of output node j; and af is the learning rate α. For threshold updating, three parameters are needed, but in order to use the same hardware structure for both weights and thresholds updating, a redundant value '1' is sent so that four parameters are needed in updating both weights and thresholds. The pseudo code for weight updating from *MicroBlaze* is shown in Figure 5.17.

   As seen in Figure 5.17, 16 parameters are sent out to FSL0 so that four weights (thresholds) can be updated. For example, updating weights between the hidden layer and output layer, the inner loop is from 0 to NODE_NUM_O. NODE_NUM_O denotes the number of nodes in the output layer which is four. This means parameters for updating four weights are sent out during each iteration of the outer loop. This is followed by the function of receiving

```
void update()
{
  //update all weights between input and hidden layer
  for(i=0;i<NODE_NUM_I;i++)
  {
    for(h=0;h<4;h++)
    {
      microblaze_nbwrite_datafsl(v[i][h],0);
      microblaze_nbwrite_datafsl(af,0);
      microblaze_nbwrite_datafsl(a[i],0);
      microblaze_nbwrite_datafsl(e[h],0);
    }
    for(h=0;h<4;h++)
    {
      microblaze_nbread_datafsl(v[i][h],1);
    }
    for(h=4;h<NODE_NUM_H;h++)
    {
      microblaze_nbwrite_datafsl(v[i][h],0);
      microblaze_nbwrite_datafsl(af,0);
      microblaze_nbwrite_datafsl(a[i],0);
      microblaze_nbwrite_datafsl(e[h],0);
    }
    for(h=4;h<NODE_NUM_H;h++)
    {
      microblaze_nbread_datafsl(v[i][h],1);
    }
  }
  //update all weights between hidden and output layer
  for(h=0;h<NODE_NUM_H;h++)
  {
    for(o=0;o<NODE_NUM_O;o++)
    {
      microblaze_nbwrite_datafsl(w[h][o],0);
      microblaze_nbwrite_datafsl(af,0);
      microblaze_nbwrite_datafsl(b[h],0);
      microblaze_nbwrite_datafsl(d[o],0);
    }
    for(o=0;o<NODE_NUM_O;o++)
    {
      microblaze_nbread_datafsl(w[h][o],1);
    }
  }
}
```

Figure 5.17: Pseudo Code for Hardware update() Function

data from the *HUM*. The parameters from the *MicroBlaze* processor are stored in the FSL0 buffer waiting to be fetched by *HUM*. After the first parameter is stored in the FSL0 buffer, *FSL0_S_Exists* switches from '0' to '1' to indicate that data is available in FSL0.

2. **From FSL0 to HUM:** The fetching of parameters from the FSL0 buffer starts when '1' is detected on port *FSL0_S_Read* at the rising clock edge. The parameters in FSL0 FIFO buffer are then moved to *FSL0_S_Data* port at a rate of one per cycle. The data is stored in the HUM module local registers. The timing of fetching data from the FSL0 channel to local registers is shown in Figure 5.18.

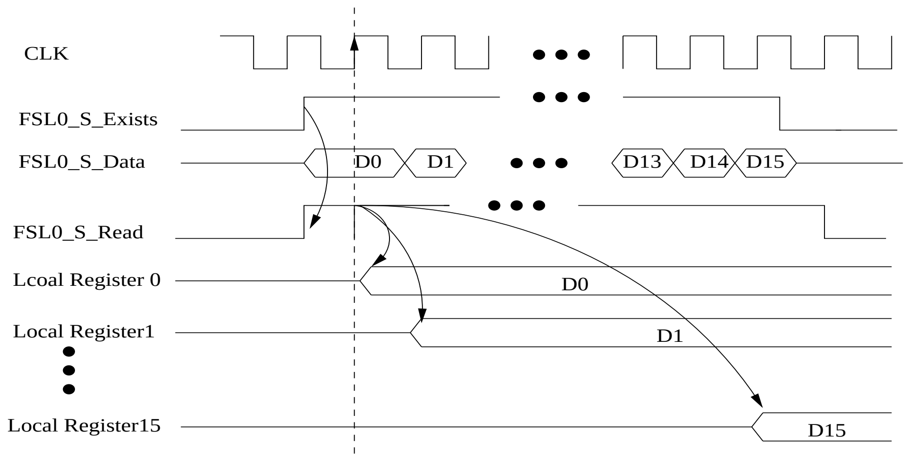

Figure 5.18: Illustration of Fetching Data from FSL0 to *HUM*

As seen in Figure 5.14 *FSL0_S_Read* is connected with *FSL0_S_Exists* to allow data to be transfered from the buffer to the FSL_S_Data port. At each rising clock edge, *FSL0_S_Exists* is also monitored by *counter1*. If it is '1', *counter1* starts to count and triggers the *storage controller* to put serial data from

FSL0_S_Data to 16 local registers. Another signal is sent from *counter1* to the *Finite State Machine* to indicate that 16 parameters have been stored locally. This signal becomes ‘1’ when *counter1* reaches the count of “17”.

5.3.2.2 Stage #2

In this processing stage, the *UPDATE UNITs* attempt to compute the updated weights (thresholds) based on the local parameters. The triggering the *UPDATE UNITs* is controlled by a *Finite State Machine*.

1. **Finite State Machine:** There are three input signals and two output signals in the state diagram shown in Figure 5.19.

   The input signals are *Ready_cal*, *Ready_out* and *Done_out*. *Ready_cal* is from *counter1*. *Ready_out* is based on “AND” logic of four “done” signals from *UPDATE UNITs* which indicate that the latter have completed their task and the updated results are ready to be sent out. The *Done_out* signal is an output signal from *counter2* module to indicate that the results have already been sent out.

   An output signal from the *Finite State Machine* is sent to the *UPDATE UNITs* for starting/stopping the latter. If *Ready_cal* is ‘1’ in “waiting” state, *Start_cal* will be ‘1’ which informs the *UPDATE UNITs* to start or continue processing. If *Ready_out* is ‘1’ in “calculating” state, this output is ‘1’ which tells *counter2* to send a signal to *Output controller* so that the updated results are sent to *FSL1_M_Data* one by one from local registers. If *Done_out* is ‘0’ in “sending” state, it will return to “waiting” state. The waveform for this process is shown in Figure 5.20.

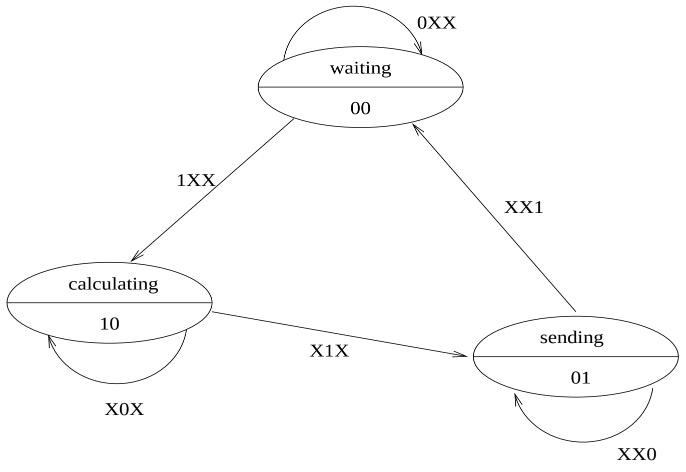

Figure 5.19: The State Transition Diagram

2. ***UPDATE UNIT:*** The task of *HUM* is achieved by four *UPDATE UNITs* as shown in Figure 5.21. There are two IEEE single precision multipliers and one IEEE single precision adder within the *UPDATE UNIT* module. For weight updating $w[h][o] = w[h][o] + \alpha b[h]d[o]$, op1 could be $w[h][o]$; op2 could be α; op3 could be $b[h]$; and op4 could be $d[o]$.

   The parameterized floating-point adder *fp_add* and multiplier *fp_mul* were introduced in Chapter 4. In order to construct an IEEE single precision adder and multiplier, two other modules were constructed (*DENORM* and *RND_NORM*). These two modules are also from Pavle's floating-point library. A detailed functional explanation of *DENORM* and *RND_NORM* are presented first and then the construction of IEEE single precision adder and multiplier are presented.

   - *DENORM*

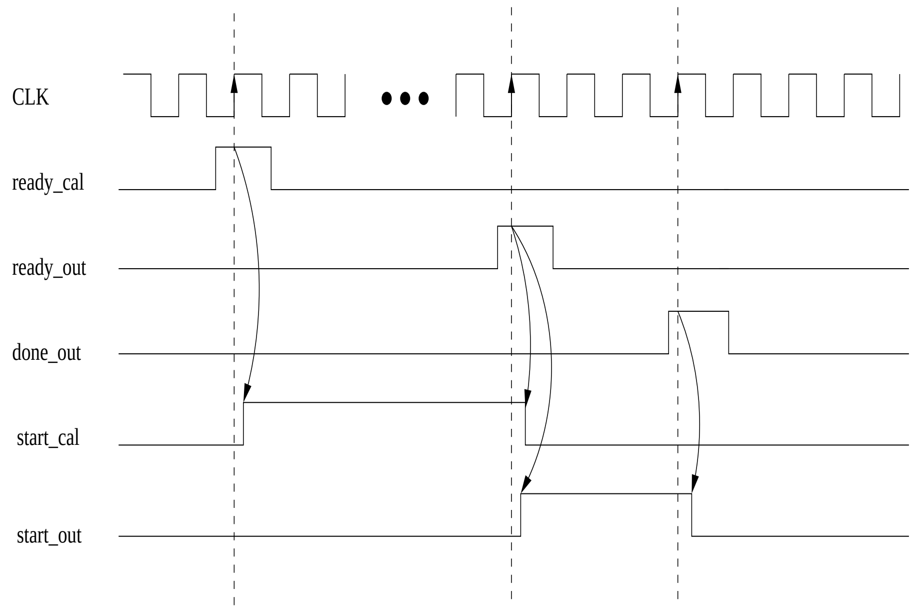

Figure 5.20: Waveform to Illustrate how FSM Works

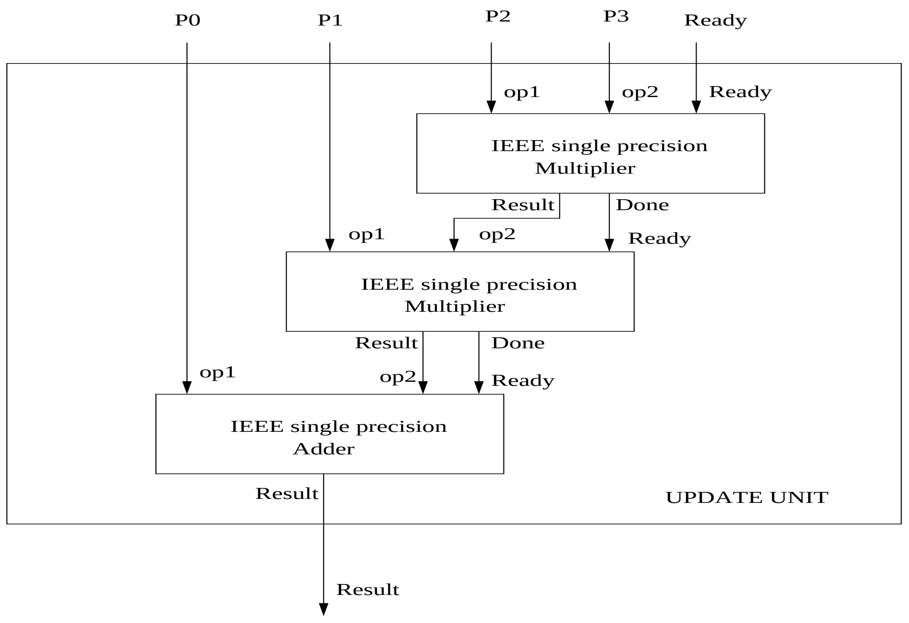
P0
P1
P2
P3
Ready
op1
op2
Ready
IEEE single precision Multiplier
Result
Done
op1
op2
Ready
IEEE single precision Multiplier
Result
Done
op1
op2
Ready
IEEE single precision Adder
Result
UPDATE UNIT
Result

Figure 5.21: Block Diagram of Update Unit

Normalization of floating-point numbers tend to make the integer part of mantissa non-zero. For example, when dealing with the binary number system, the only non-zero value is '1'. For a normalized floating-point number, the '1' in the integer part of the mantissa is redundant and only the fractional part of the mantissa is stored. However, IEEE standard 754 has the following rules: when the floating-point number is not zero, it is stored as a normalized format, and, when the floating-point number is zero, the integer part of the mantissa is zero. Therefore, the *DENORM* module is introduced to add the implied '1' or '0' to the mantissa to form a de-normalized floating-point number based on the rules of IEEE standard 754.

- *RND_NORM*

  Following any floating-point operation, results need to be converted to their normalized format. This is achieved by *RND_NORM* module shown in Figure 5.22. There are two sub-modules: *NORMALIZER* and *ROUND_ADD* included in *RND_NORM*. The function of *NORMALIZER* is to convert a floating-point number into a normalized form. This is accomplished by shifting the mantissa to the left until the MSB is '1' while decreasing the exponent for every bit by which the mantissa is shifted. The function of *ROUND_ADD* is to reduce the mantissa bit-width to its original size. For example, floating-point addition introduces a guard digit which make the mantissa bit-width increase by 1 bit. The result must be rounded to the same length as the original mantissa. Two rounding modes can be chosen in *ROUND_ADD* which are *round to zero* (truncating) and *round to nearest* (which is used in the developed HUM module).

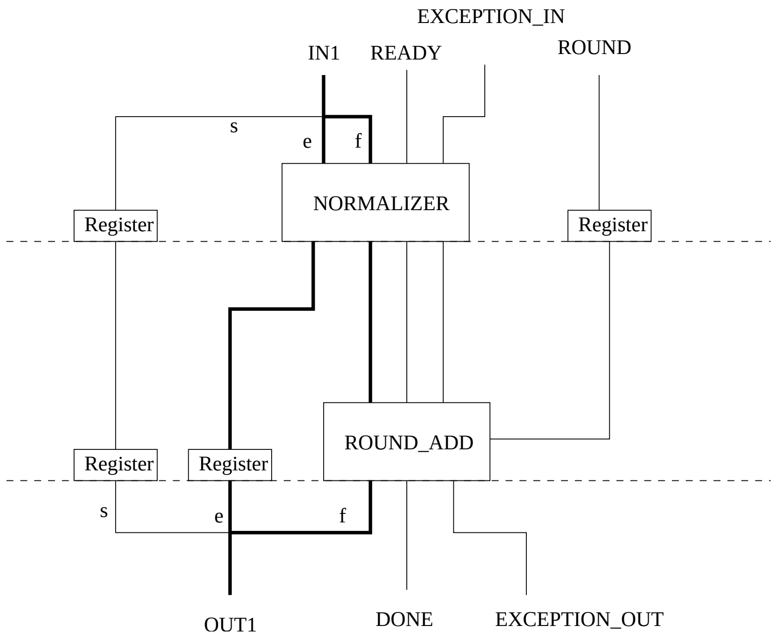

Figure 5.22: Rounding and Normalizing

- *IEEE single precision adder*

  The IEEE single precision adder is constructed from three modules: *fp_add*, *denorm* and *rnd_norm*. *fp_add* was introduced in Chapter 4 and the remaining two modules were introduced in the previous sub-section. From Figure 5.23 it is clear that *denorm* is used twice to add the implied integer bit into both input operands. *fp_add* executes floating-point addition on these two prepared operands. Due to the *guard bit* introduced in addition and *denorm*, the mantissa is 25 bits wide. *rnd_norm* is finally used to convert the result back to the IEEE single precision format.

- *IEEE single precision multiplier*

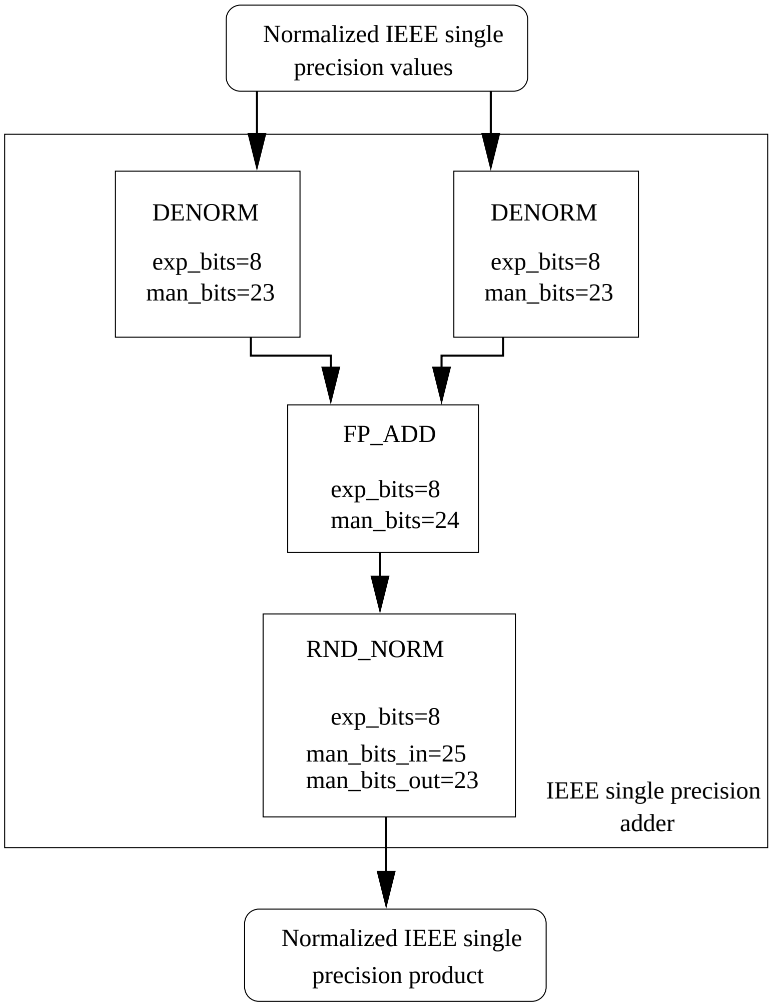

Figure 5.23: IEEE Single Precision Adder

The IEEE single precision multiplier is constructed from three modules: *fp_mul*, *denorm* and *rnd_norm*. *fp_mul* was also introduced in Chapter 4. From Figure 5.24 it can be seen that *denorm* is used twice to add the implied integer bit into both input operands. *fp_mul* executes floating-point multiplication on these two prepared operands. After multiplication, the length of the mantissa is twice that of the input operands which is 48 bits. *rnd_norm* is finally used to convert the result back to IEEE single precision format.

5.3.2.3 Stage #3

In this stage, four updated weights (thresholds) are directed to the *FSL1_M_Data* port from local registers at one per cycle mode. The *FSL1_M_Write* signal is set to '1' during the sending state. When the *MicroBlaze* module detects the high signal at the rising edge of the clock, the data on *FSL1_M_Data* port are withdrawn.

5.3.3 Results

There are two parts to the result. Firstly the FPGA usage of *HUM* function, and FSL channel is presented. Secondly, the time consumed by *HUM* is presented. Compared with the pure *MicroBlaze* solution introduced earlier, the speedup to the *update* function has an overall gain in performance calculated by Amdahl's Law.

5.3.3.1 FPGA usage

As indicated in Figure 5.13, the system includes both a *MicroBlaze* with peripherals and *HUM* with FSL channel. The FPGA usage of a *MicroBlaze* with peripherals

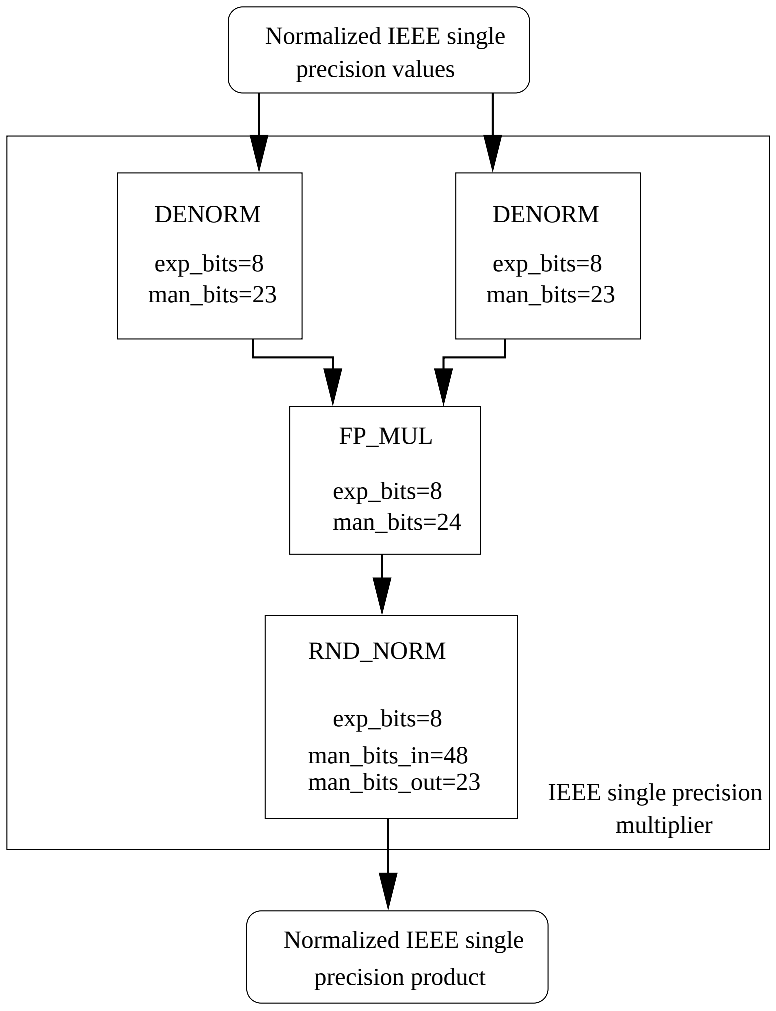

Figure 5.24: IEEE Single Precision Multiplier Using Pavle's Library

is shown in Table 5.2 and the FPGA usage of *HUM* with FSL is shown in Table 5.5. If the number of slices is used to count the FPGA area occupied, the whole system consumes 49% of the total area of a Virtex-II XC2V2000 device.

	MULTs	16-bit mem.	Fun. Gen.	Carry Sym.	Flip Flops
h_update	32	8	5795	1489	3868
fsl	0	34	10	5	6

Table 5.5: The FPGA Usage on *HUM*

5.3.4 Amdahl's law [Henn95]

Amdahl's Law measures the performance gain by using some of the faster components of a computer as follows:

1. *The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement* - For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call $Fraction_{enhanced}$, is always less than or equal to 1.

2. *The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program.* This value is the time of the original mode over the time of the enhanced mode: If the enhanced mode takes 2 seconds for some portion of the program that can completely use the mode, while the original mode took 5 seconds for

the same portion, the improvement is 5/2. We will call this value, which is always greater than 1, $Speedup_{enhanced}$.

The execution time using the original machine with the enhanced mode will be the time spent using the nonenhanced portion of the machine plus the time spent using the enhancement:

$$Execution\ time_{new} = Execution\ time_{old} \times ((1 - Fraction_{enhanced}) + \frac{Frac_{enhanced}}{Speedup_{enhanced}}) \tag{5.1}$$

The overall speedup is the ratio of the execution times:

$$Speedup_{overall} = \frac{Execution\ time_{old}}{Execution\ time_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \tag{5.2}$$

5.3.4.1 The Time consumed by ***HUM***

The time consumed by the dedicated hardware *update* module is shown in Table 5.6.

% time	Cycles	Seconds	Name
92.2	2077492	0.0207	forward
1.4	31283	0.0003	backward
6.4	145196	0.0014	hardware update

Table 5.6: Profiling on Co-implement System

As can be seen in Table 5.6, *update()* function implemented by hardware only requires 145196 cycles whereas *MicroBlaze* requires 1730347 cycles. In equation 5.2, $Speedup_{enhanced} = \frac{1730347}{145196} = 11.91$ and $Fraction_{enhanced}$ is determined by Table 5.4 where $Fraction_{enhanced}$ is 45.07% (45.07% is the fraction of function *update* to be enhanced). The overall speedup achieved over a pure software implementation is is around 1.69 (Amdahl's Law) on average.

5.4 Summary

In this Chapter, two systems were proposed for solving the face recognition problem. The first contains a *MicroBlaze* soft processor core implemented onto an FPGA. The second system contains both a *MicroBlaze* and a dedicated hardware module. The results show that the *update()* function implemented by hardware is over 10 times faster than the software implementation. However, the overall speedup achieved is approximately 1.69 over a pure software implementation.

Chapter 6

Conclusion

Artificial Neural Networks have been proven to be of broad applicability to real world problems such as pattern recognition. However, the inherent parallel distributed character of ANN cannot be implemented efficiently by a general processor. Many efforts have been made in the past to implement ANNs onto hardware such as ASICs and FPGAs to achieve speedup [Eldr94, Hika99, JB98, Ferr94, Mart94]. This dissertation attempts to find the means of balancing flexibility and performance to solve the problem of mapping a large ANN architecture onto a single FPGA chip. For these cases, a Hardware/Software Co-design technique is used to design a system containing both processor(s) and dedicated hardware.

The effect of arithmetic representation formats on implementing ANN on FPGAs was investigated. The main focus was to examine the tradeoffs between precision and range of various formats that can target FPGA resources. The basic ANN processing elements include multiplication and addition operations. As such, twenty-five Floating and Fixed formats were tested where a multiplier and an adder were implemented on an FPGA and their area requirements compared. Results obtained already indicate that floating-point formats offer more area-efficient implementation without penalty on precision or range. The results also show that the target FPGA device could have a major impact on the resources required. However, implementing a large ANN onto only one FPGA is still not feasible today. In order to balance performance and flexibility, a Hardware/Software Co-design approach should be followed.

In the second part of the thesis, two systems were proposed to solve the face recognition problem by an MLP-BP ANN. The first system consisted of a *MicroBlaze* soft processor core while the second system included both a *MicroBlaze* processor and a dedicated hardware module. The results show that the subfunction implemented by hardware is over 10 times faster than running on the *MicroBlaze*. However, the overall speedup achieved is approximately 41% over a pure software implementation. The system containing both hardware and software has a good balance between flexibility and performance.

6.1 Future Work

Here are several avenues for future research. If performance is the main concern, an ANN can be implemented onto FPGAs by only programmable logic. The work in this case could focus on two issues: run-time reconfiguration so that a large ANN can be mapped on only one FPGA chip; the design of more efficient adder, multiplier, and sigmoid functions which are basic arithmetic operators of ANNs is of importance. If performance and flexibility are going to be better balanced, a system containing more *MicroBlazes* and more dedicated hardware modules could be a better solution. Also, if only arithmetic operators are implemented onto hardware and the scheduling task is implemented by a *MicroBlaze*, a better performance can be achieved without losing any flexibility. The limitation of this work is that no real-life benchmarks were tested on the proposed systems and therefore, more real-life benchmarks should be tested.

Appendix A

Glossary

ANN	:	Artificial Neural Network
EDK	:	Embedded Development Kit
BP	:	BackPropagation
MLP-BP	:	Multi-Layer Perceptron-BackPropagation
FPGA	:	Field Programmable Gate Array
FSL	:	Fast Simplex Link
BRAM	:	Block RAM
IP	:	Intellectual Property
OPB	:	On-Chip Peripheral Bus
LMB	:	Local Memory Bus
RC	:	Reconfigurable Computing
RTL	:	Register Transfer Logic
PCA	:	Principal Component Analysis
LDA	:	Linear Discriminant Analysis

DCM	: Digital Clock Manager
VHDL	: Very High Speed Integrated Circuit Hardware Description Language
LUT	: Lookup Table
ZBT	: Zero Bus Turnaround
JTAG	: Joint Test Action Group
UART	: Universal Asynchronous Receiver/Transmitter
VLSI	: Very Large Scale Integration
HUM	: Hardware Update Module

Appendix B

Embedded Development Kit

All experiments in this thesis were carried out using Xilinx EDK6.1. This kit contains a rich set of design tools and wide selection of standard peripherals required to build embedded processor systems using *MicroBlaze*, the industry's fastest soft processor solution. Embedded System Tools (EST) is included in the EDK which consists processor platform tailoring utilities, software application development tool, a full featured debugging tool chain, device drivers and libraries. In order to help reader have a top view on how to design an Embedded System by EST, the architecture of EST is introduced here. The detail information on each tool inside of EST can be found in Xilinx Embedded System Tools Guide [httpk]

- **Embedded Software Tool Architecture[httpk]** Figure B.1 depicts the EST architecture (XPS is Xilinx Platform Stdio). Multiple tools based on a common framework allow users to design a complete embedded system. System design consists of the creation of the hardware and software components of the embedded processor system, and optionally, a verification or simulation component as well. The hardware component consists of an automatically generated hardware platform that can be optionally extended to include other hardware functionality specified by the user.

The software component of the design consists of the software platform generated by the tools, along with the user designed application software. The verification component consists of automatically generated simulation models targeted to a specific simulator, based on the hardware and software components.

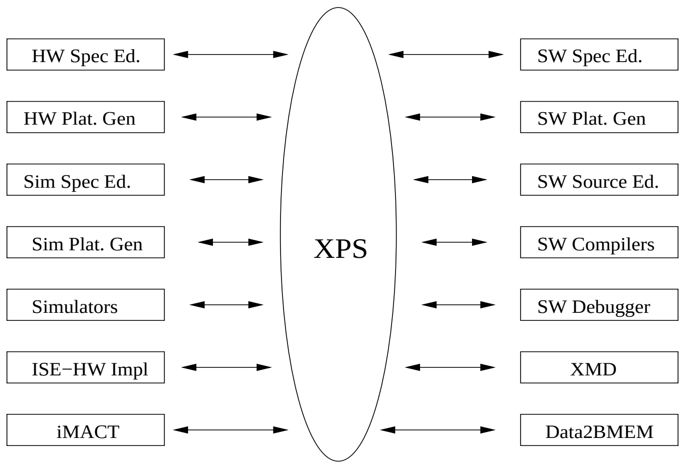

Figure B.1: Embedded Software Tool Architecture

- **Tool Flows**

  A typical embedded system design project involves the following phases:

  – hardware platform creation,

  – hardware platform verification (simulation),

  – software platform creation,

– software application creation, and

– software verification (debugging).

Hardware Platform Creation The hardware platform creation is depicted in Figure B.2. The hardware platform consists of one or more processors and peripherals connected to the processor buses. User can define their own peripherals or invoke from libraries provided by Xilinx. The system architecture, peripherals and embedded processors are defined by MHS (Microprocessor Hardware Specification) file. The MHS file also defines the connectivity of the system, the address map of each peripheral in the system and configurable options for each peripheral.

The Platform Generator tool (platgen) creates the hardware platform using the MHS file as input. Platgen creates netlist files in various formats (NGC, EDIF), as well as support files for downstream tools, and top level HDL wrappers to allow users to add other components to the automatically generated hardware platform.

Verification Platform Creation The verification platform is based on the hardware platform. The verification specification allows the user to specify a simulation model for each processor, peripheral or other module in the hardware platform. The MHS file is processed by the Simgen tool to create simulation files (VHDL, Verilog or various compiled models) along with some command files for specific simulators supported by the tool. The entire process of generating the verification platform is depicted in Figure B.3.

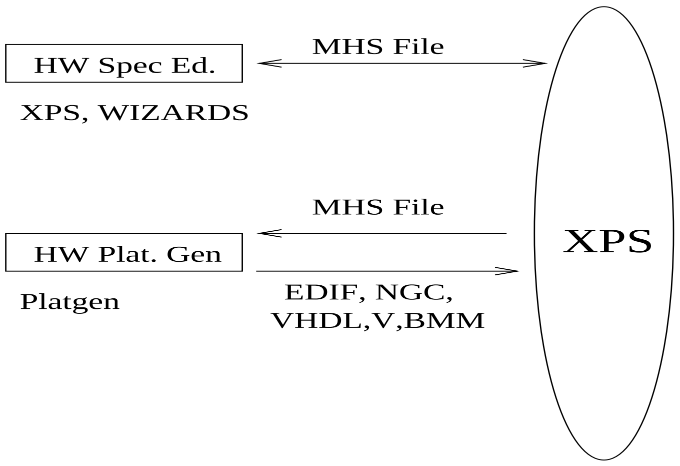

Figure B.2: Hardware Platform Creation

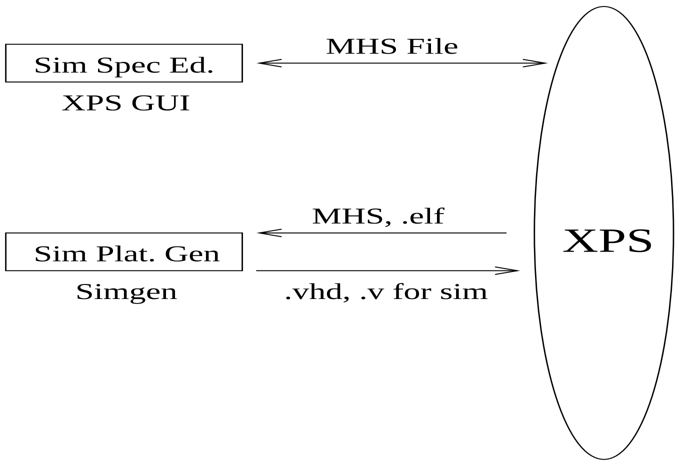

Figure B.3: Verification Platform Creation

Software Platform Creation The software platform is defined by the MSS (Microprocessor Software Specification) file. The MSS file defines driver and library customization parameters for peripherals, processor customization parameters, standard input/output devices, interrupt handler routines, and other related software features. The MSS file is an input to the Library Generator tool (LibGen) for customization of drivers, libraries and interrupt handlers. The entire process of creating the software platform is shown in Figure B.4.

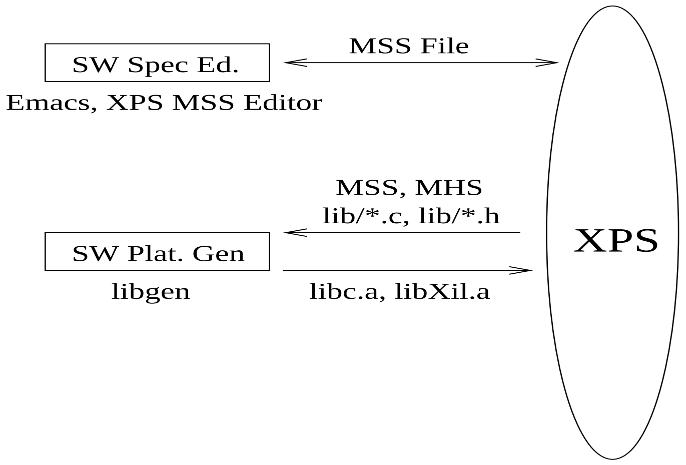

Figure B.4: Software Platform

Software Application Creation and Verification The software application is the code that runs on the hardware and software platforms. The source code for the application is written in a high level language such as C or C++, or in assembly language. Once the source files are created, they are compiled and linked to generate executable files in the ELF (Executable and Link Format) format. XMD and the GNU debugger

(GDB) are used together to debug the software application. XMD provide an instruction set simulator, and optionally connects to a working hardware platform to allow GDB to run the user application. The entire process is depicted in Figure B.5.

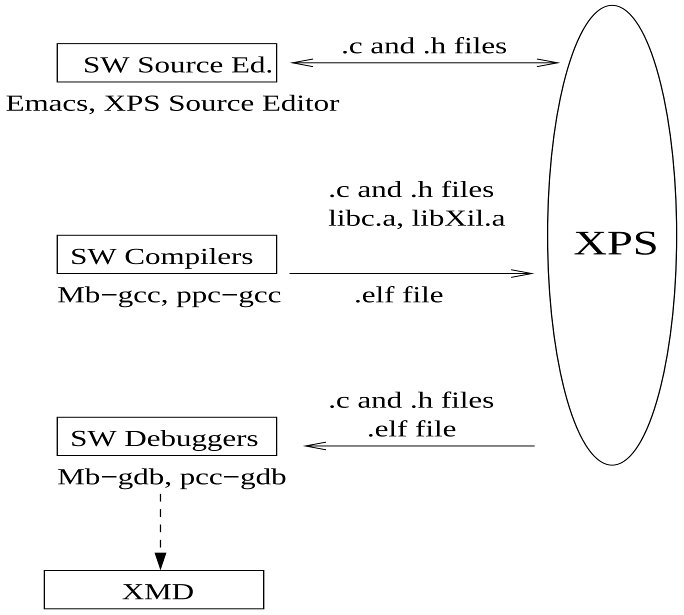

Figure B.5: Software Application Creation and Verification

Appendix C

Xilinx MicroBlaze Soft Processor Core [Micr]

MicroBlaze is a full Harvard, RISC pipelined architecture with 32-bit data and 32-bit instruction words. It supports a subset of IBM's COREConnect On-chip Peripheral Bus (OPB) architecture [httpf]. It also contains a Local Memory Bus (LMB) for fast access to local BRAM for both Instruction and Data. The *MicroBlaze* pipeline is a parallel pipeline, divided into three stages: Fetch, Decode and Execute. All instructions take one clock cycle except for the following:

- Load/Store: 2 clock cycles+I/O latency
- Multiply: 3 clock cycles
- Branches: 3 clock cycles

The features of this soft processor core are the following:

- Supports Virtex, Virtex-E, Virtex-II Pro, Spartan-II, and Spartan-IIE devices

- Performance: 102 Dhrystone MIPS (D-MIPS) on Virtex-II Pro device at 150 MHz
- Minimum logic requirements: 900 logic cells
- 32-bit pipelined RISC architecture
- 32×32-bit general purpose registers
- Implementation in Virtex-II and later device support hardware multiply
- Supports Local Memory Bus (LMB) for fast access of on-chip BRAMs
- Supports IBM CoreConnect On-chip Peripheral Bus (OPB) for accessing peripherals
- Processor peripherals compatible with PowerPC on Virtex-II Pro
- Complete hardware and software development tool and debug solution

An embedded system built around *MicroBlaze* is comprised of the following:

- *MicroBlaze* Soft Processor Core
- On-chip Local Memory
- Standard Bus Interconnects
- On-chip Peripheral Bus (OPB) Peripherals

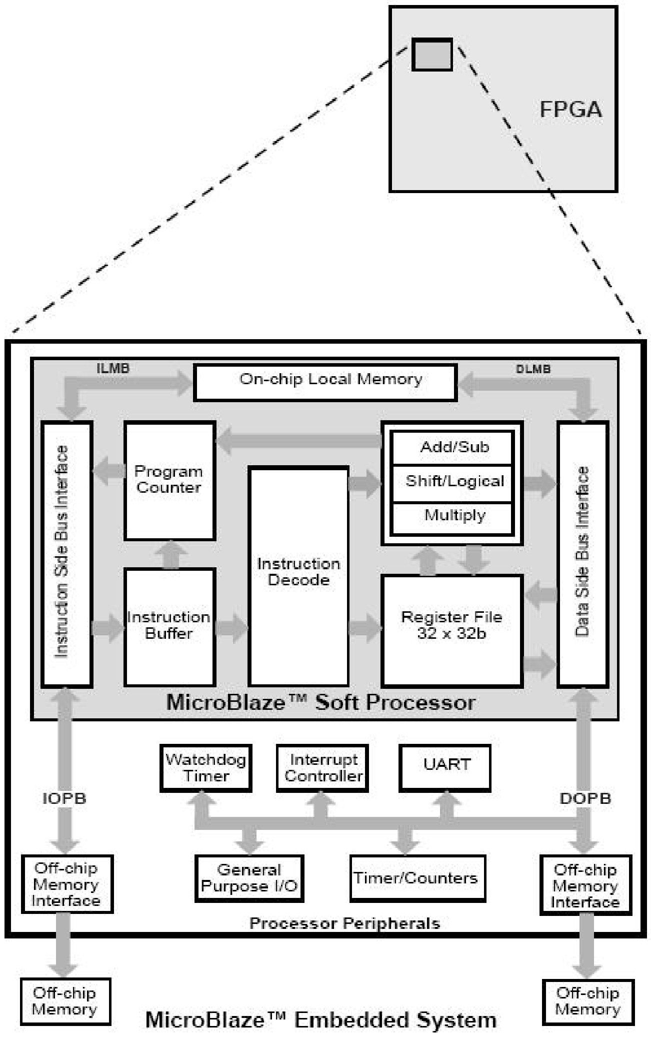

Figure C.1: MicroBlaze RISC 32-Bit Soft Processor System Interconnect Diagram

Appendix D

Rapid prototyping board

The final design is implemented on Xilinx Virtex-II Multimedia board shown in Figure D.1. As can be seen, in the center of the board is a Xilinx Virtex-II XC2V2000 FPGA chip. The five chips around it are 512 $\times$ 36bit 130MHz ZBT (Zero Bus Turnaround) RAM [httpg]. The left side of the board contains a 16M Flash memory and a serial port. The up and right side of the board are some audio and video I/O ports. The botton of the board are several push buttons. This board is designed as a compact platform for developing multimedia applications. The Multimedia board uses a Virtex-II XC2V2000-FF896 as the user application FPGA. Xilinx MicroBlaze 32-bit soft processor can also be used on the Multimedia board as a processing engine. The board supports five independent banks of 512K x 36bit 130MHz ZBT RAM with byte write capability. This memory may be used as microprocessor code/data store, or as video frame buffers.

Real time video is supported with a PAL - NTSC video decoder, encoder pair. The video may originate in either S-video or composite video format and is converted into CCIR 601/CCIR656 8-bit or 10-bit extended format for processing by the FPGA. The Virtex-II FPGA also generates the digital video stream for the video encoder. Video output is provided in S-video, composite video as well as RGB formats. A triple 8 bit DAC is also included to support SVGA output up to 1024 x 768 resolution with 85Hz refresh.

Audio processing is also supported with stereo line level inputs and outputs, a headphone jack and microphone input.

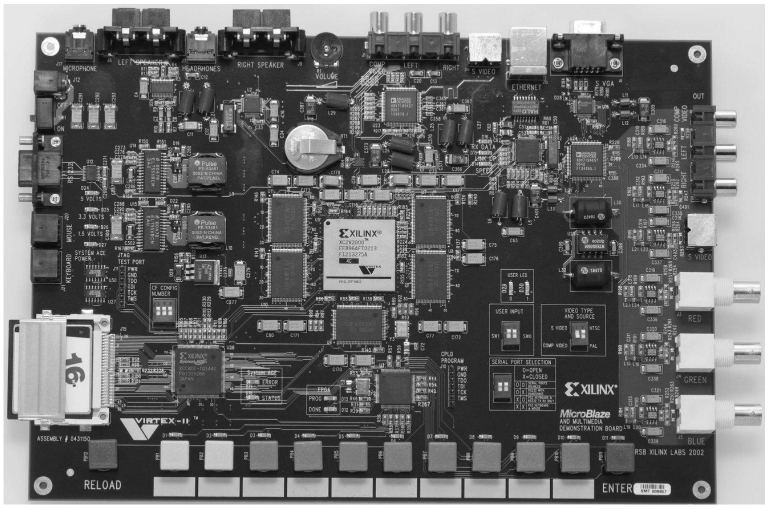

Figure D.1: Xilinx Multimedia Development Board

Multimedia Board Features:

- Virtex-II XCV2000-FF896
- Audio CODEC compliant with AC97 and stereo amplifier with 18-bit sigma-delta A/Ds and D/As
- Supports a single channel of real time PAL or NTSC video input from in either composite or S-video (Y/C) format Supports a single channel of real

time PAL or NTSC video output composite S-Video (Y/C) format and RGB simultaneously

- SVGA output
- Supports five independent banks of 512K x 36bit 130MHz ZBT RAM with byte write capability
- Onboard network connection ,10/100 Ethernet, with a unique MAC address assigned to each board
- RS232 port
- Button cell battery to support FPGA configuration data encryption.
- Embedded SystemACE controller for high-speed FPGA configuration.
  - The user application FPGA has access to the SystemACE microprocessor port to allow the Compact Flash to be used as a file storage media.
- Keyboard and mouse
- Integrated power supplies
- Speaker output with adjustable volume
- Headphone and microphone
- 10 pushbuttons as user inputs to the board
- Two DIP switches with two LEDs for visual feedback.

Appendix E

Computing Platforms

The existing computing platforms include processors (general processors and Digital Signal Processing (DSP) processors), Reconfigurable Computing (RC) platform, Application Specific Integrated Circuits (ASIC) and hybrid systems where both hardware and software (embedded processor) are implemented together.

Processors are extensively used in scientific computations. The processors have fixed components such as ALU (Arithmetic Logic Unit) and Control Logic. Applications are executed by decoding a stream of instructions and operating on data stored in the memory. Different applications can be realized by easily coding different instruction stream and running it on processors (software solution). However, instruction fetching, decoding, executing and memory accessing are sequential procedures. The sequential nature of processors could be a bottle-neck in real-time applications.

ASICs are widely used in real-time applications. It is designed on integrated circuit (hardware solution) for a specific application and hence, each ASIC has fixed functionality but high performance by extensively using parallelism and pipelining for a specific application. The disadvantage of ASICs is the restriction of function modification to the algorithms implemented on them after fabrication.

RC gives you a solution in compromise between flexibility (processor) and performance (ASIC). FPGA is usually a kind of reconfigurable medium which can be configured at compile-time or run-time to facilitate greater flexibility without compromising performance. The performance is also achieved by parallelism and pipelining implementation of the algorithms. Compared to conventional processors, RC solution has significant performance advantages. Complex algorithms can be mapped onto FPGAs to achieve higher computation density than processors, because the logic for instruction fetching, decoding, executing is not necessary and parallelism and pipelining techniques can be used. Compared to ASIC, RC solution has more flexibility due to the reconfigurable ability. However, the performance can only be close to ASICs in terms of speed.

In addition to RC medium, DSP is another platform that can balance flexibility and performance. In this kind of platform, some functions realized by ALU in a general processor are moved to dedicated hardware like multipliers, accumulators, etc. which are frequently used in DSP applications. In other words, DSP processors are special processors dedicated for DSP applications.

Today, the capacity of FPGAs has reached a point that both processors and dedicated hardware can be implemented at the same time on a single FPGA chip. For example, *MicroBlaze* soft processor IP core requires approximately 900 LUTs for Virtex-II FPGAs and Virtex-II xc2v2000 FPGA (The one used in this thesis) has 21,504 LUTs. A system on a FPGA including both processor like *MicroBlaze* and other hardware modules can be also used to balance flexibility and performance. This hybrid system is considered as a embedded computing system.

Figure E.1 better represents the relation between these platforms discussed above.

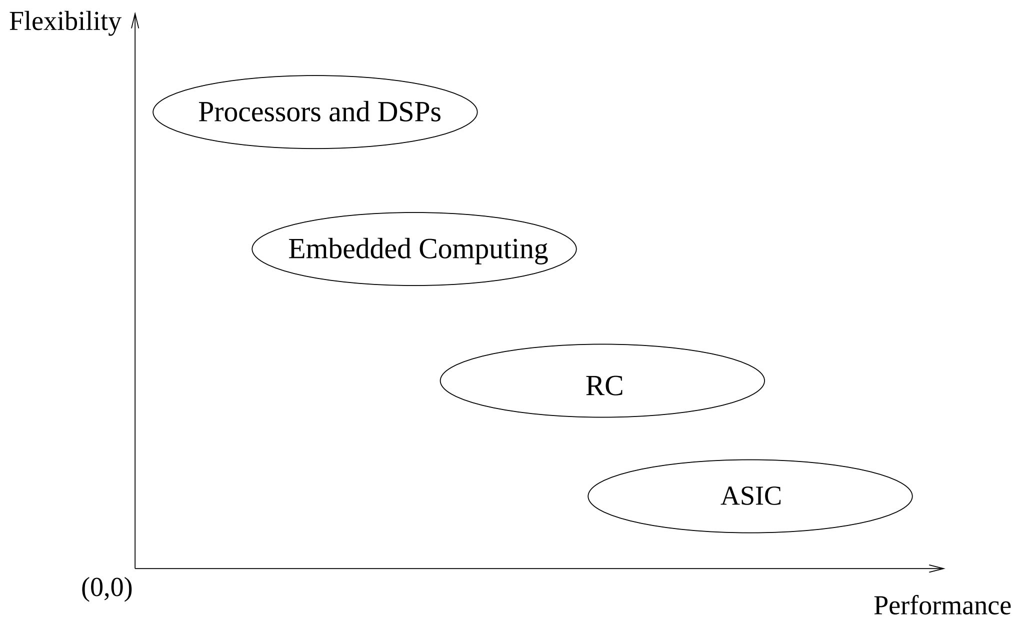

Figure E.1: Comparison Between Different Platform in Terms of Performance and Flexibility

Appendix F

MicroBlaze IP-Cores

F.1 *MicroBlaze*

Xilinx *MicroBlaze* soft processor IP core can be used in designing an *Embedded System* in Xilinx FPGAs, which may include one or more *MicroBlazes* with several peripherals where are connected through OPB [httpi] bus.

The *MicroBlaze* is the industry's fastest soft processing solution [httpl]. It's a 32-bit RISC consisting a three stage pipeline with dedicated instructions and data paths (Harvard style) shown in Figure F.1.

The MicroBlaze embedded soft core includes the following features:

1. Thirty-two 32-bit instruction word with three operands and two addressing modes

2. Separate 32-bit instruction and data buses that conform to IBM's OPB (On-chip Peripheral Bus) specification

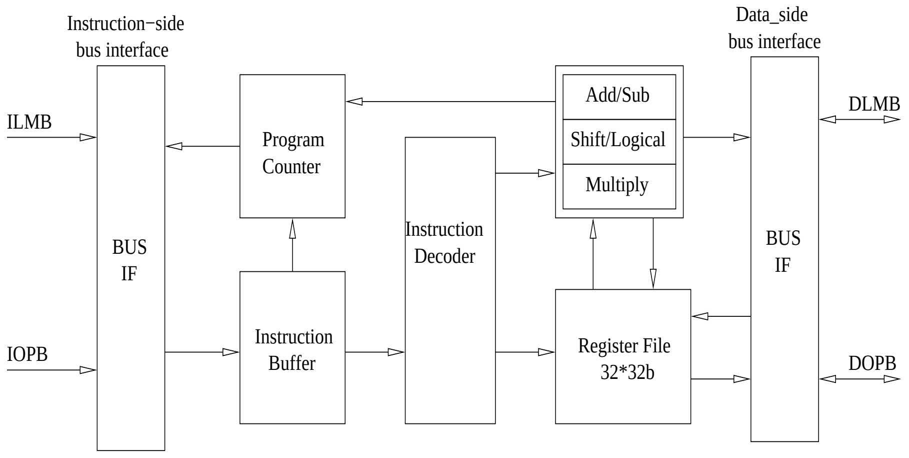

Figure F.1: MicroBlaze Core Block Diagram

3. Separate 32-bit instruction and data buses with direct connection to on-chip block RAM through a LMB (Local Memory Bus)

4. 32-bit address bus

5. Single issue pipeline

6. Hardware multiplier (in Virtex-II and subsequent devices)

F.2 Xilinx *MicroBlaze* bus interfaces

The interfaces used in Figure F.1 are: OPB[httpi] which stands for On-chip Peripheral Buses and LMB [httpi] which stands for Local Memory Buses. In Figure F.1, IOPB and ILMB are instruction path whereas DOPB and DLMB are data path.

- **Local Memory Bus (LMB)** The LMB is a highly efficient bus for Xilinx Block RAM accessing. LMB provides guaranteed performance of 125 MHz for local memory subsystem.

- **On-chip Peripheral Bus (OPB)** The OPB can be used to connect to large external memory either instruction or data memory. In addition, peripherals which are compatible with *MicroBlaze* can also be connected to *MicroBlaze* through this bus.

- ***MicroBlaze* Bus Configurations** There are six configurations you can use depending on code size and data spaces and whether you require fast access to internal Block RAM. The six different configurations are shown in Figure F.2.

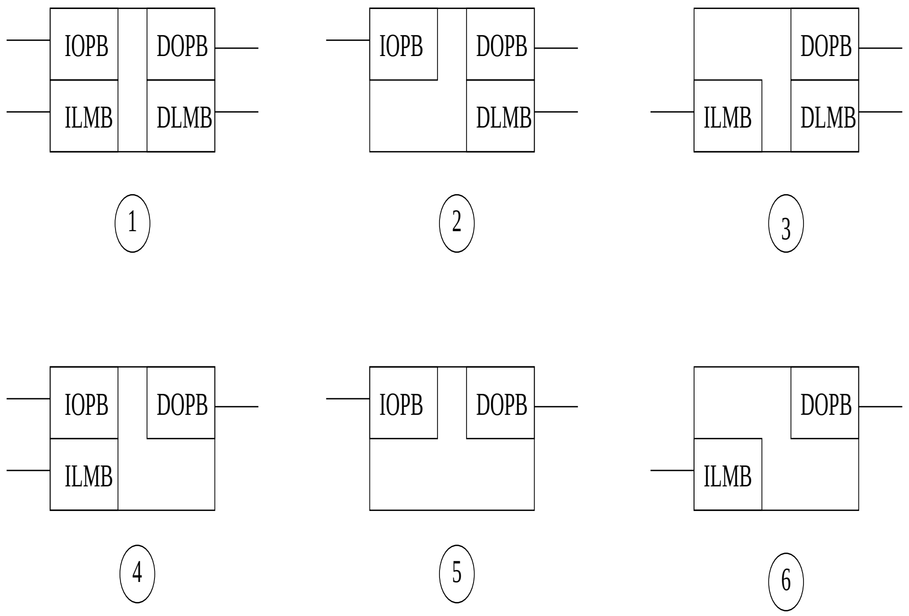

Figure F.2: *MicroBlaze* Bus Configurations

The system shown in Figure 5.12 is in configuration No. 3. This configuration allows the CPU core to operate at the maximum clock rate due to the simpler instruction-side bus structure. The instruction-side LMB provides two-cycle pipelined read-access from the Block RAM for an effective access rate of one instruction per clock. Since the objective of this thesis is to compare the software implementation with hybrid implementation of *backpropagation* algorithm on FPGA in terms of speed and flexibility, 4 images is used in the training process for simplicity. This will allow less memory usage where only Block RAM inside FPGA is enough. All the instruction code and image data are stored in Block RAM of XC2V2000 FPGA on Xilinx Multimedia board where 1008K bit Block RAM is available.

F.3 OPB Block RAM[httpi]

The *OPB Block RAM* is a module that is going to be attached to the OPB. The features of *OPB Block RAM* are following:

Features

- OPB V2.0 bus interface with byte-enable support
- Number of Block RAMs is configurable
- Handles byte, half-word and word transfers
- Other port of the BRAM is available for customer designs
- Handles Virtex, Virtex-E, Spartan-II and Virtex-II PRO type of Block RAM

To allow you to obtain an *OPB Block RAM* that is uniquely tailored for your system, certain features can be parameterized in the *OPB BRAM* design. This allows you to configure a design that only utilizes the resource required by your system, and operates with the best possible performance.

F.4 OPB Block RAM Controller

The Block RAM is connected to *MicroBlaze* processor through OPB Block RAM Controller. Two parameters can be set to determine how much Block RAM can be addressed. For example, if the *Base Address* is 0x00000000 and the *High Address* is 0x0000ffff, 32 blocks (2K Bytes each) of Block RAM can be addressed.

F.5 OPB JTAG_UART[httpi]

In the 1980s, the Joint Test Action Group (JTAG) developed a specification for boundary-scan testing that was standardized in 1990 as the IEEE Std. 1149.1-1990. In 1993 a new revision to the IEEE Std. 1149.1 standard was introduced (titled 1149.1a) and it contained many clarifications, corrections, and enhancements. In 1994, a supplement that contains a description of the boundary-scan Description Language (BSDL) was added to the standard. Since that time, this standard has been adopted by major electronics companies all over the world. Applications are found in high volume, high-end consumer products, telecommunication products, defense systems, computers, peripherals, and avionics. Now, due to its economic advantages, smaller companies that cannot afford expensive in-circuit testers are using boundary-scan.

While it is obvious that boundary-scan based testing can be used in the production phase of a product, new developments and applications of the IEEE-1149.1 standard have enabled the use of boundary-scan in many other product life cycle phases. Specifically, boundary-scan technology is now applied to product design, prototype debugging and field service. *OPB JTAG_UART* is used here for debugging the system by Xilinx Microprocessor Debugger (XMD) [httpj]. When you activate XMD tool and type *"mbconnect stub"*, XMD will automatically detect the JTAG cable, chain and FPGA device containing the *MicroBlaze* system, and connect to the JTAG core of the system. This JTAG core can be used later on for debugging the software running on *MicroBlaze*.

F.6 OPB UART Lite[httpi]

A UART (universal asynchronous receiver / transmitter) is responsible for performing the main task in serial communications with computers. The device changes incoming parallel information to serial data which can be sent on a communication line. A second UART can be used to receive the information.

OPB UART Lite is used for printing some results to the *hyper-terminal* serial communication tool on PC side which is connected to the *MicroBlaze core* system through serial port and also the testing image data could be transmitted from PC to *MicroBlaze core* system through *OPB UART Lite*. It has the following features:

Feature

- OPB V2.0 bus interface with byte-enable support

- Supports 8-bit bus interface

- One transmit and one receive channel (full duplex)
- 16-character transmit FIFO and 16-character receive FIFO
- Number of data bits in a character is configurable (5-8)
- Parity; can be configured for odd or even
- Configurable baud rate

F.7 OPB Timer/Counter[httpi]

The **TC** (Timer/Counter) is a 32-bit timer module that attaches to the OPB (On-Chip Peripheral Bus).The **TC** is organized as two identical timer modules. Each timer module has an associated register (the Load Register) that is used to hold either the initial value for the counter for event generation or a capture value, depending on the mode of the timer. The **TC** block diagram is shown in the Figure F.3.

Timer Modes

There are three modes can be used in TC. They are Generate Mode, Capture Mode or Pulse Width Modulation (PWM) Mode with the two timer/counter modules.

- **Generate Mode**

  In Generate Mode, the value in the Load Register is loaded into the counter and the counter begins to count (upward or downward by the UDT bit in TCSR) when it is enabled. On transition of the carry out of the counter,

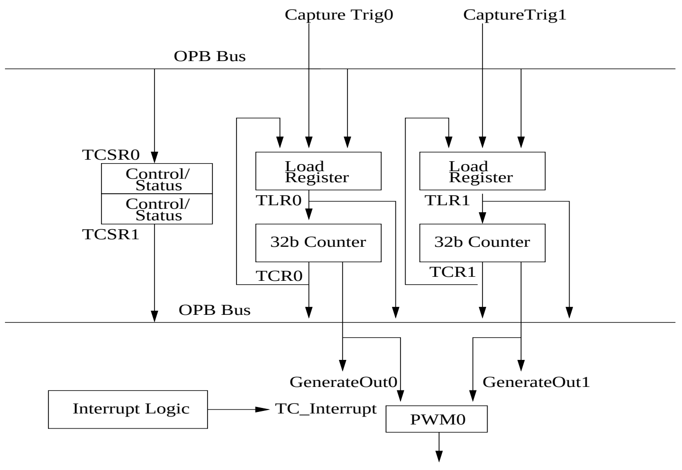

Figure F.3: Timer/Counter Organization

the counter stops or automatically reloads the generate value from the Load Register and continues counting (selectable by the ARHT bit in TCSR). The TINT bit is set in TCSR and, if enabled, the external GenerateOut signal is driven to 1 for one clock cycle. If enabled, the interrupt signal for the timer is driven to 1 for one clock cycle.

- **Capture Mode**

  In Capture Mode, the value of the counter is stored in the Load Register when the external capture signal is asserted. The TINT bit is also set in TCSR on detection of the capture event. The counter can be configured as an up or down counter for this mode (selectable by the UDT bit in TCSR). The ARHT bit controls whether the capture value is overwritten with a new capture value before the previous TINT flag is cleared. This mode is useful for timer tagging external events while simultaneously generating an interrupt.

- **Pulse Width Modulation (PWN) Mode**

  In PWN mode, two timer/counters are used as a pair to produce an output signal (PWN0) with a specified frequency and duty factor. Timer0 sets the period and Timer1 sets the high time for the PWN0 output.

F.8 OPB GPIO[httpi]

GPIO stands for General Purpose Input and Output. **OPB GPIO** is the core which can be used to connect the input/output components like LEDs and switches on the development board. The features are following:

- OPB V2.0 bus interface with byte-enable support
- Support 32-bit, 16-bit and 8-bit bus interface
- Each GPIO bit dynamically programmable as input or output
- Number of GPIO bits configurable up to size of data bus interface
- Can be configured as inputs-only to reduce resource utilization

Bibliography

[ANSI85] New York ANSI/IEEE, "Ieee standard ofr binary floating point arithmetic, std 754-1985 edition," 1985.

[Auda99] G. Auda and M. Kamel, "Modular neural networks," *International Journal of Neural Networks*, vol. 9, pp. 129–151, April 1999.

[Bela02] Pavle Belanovic, "Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application," M.A.Sc Thesis, ECE Department, 2002.

[Blum98] M. Blumenstein and B. Verma, "A Neural Based Segmentation and Recognition Technique for Handwritten Words," *World Congress on Computational Intelligence (WCCI '98)*, 1998.

[Brun93] R. Brunelli and T. Poggio, "Face Recognition: Features versus Templates," pp. 1042–1052, 1993.

[BV98] M. Blumenstein B. Verma and S. Kukarni, "Recent Achievements in Off-line Handwriting Recognition Systems," *International Conference on Computational Intelligence and Multimedia Applications (ICCIMA '98)*, 1998.

[CBus93] C.Busch and M.H.Gross, "Interactive Neural Network Texture Analysis and Visualization for Surface Reconstruction in Medical Imaging," *EUROGRAPHICS '93, Computer Graphics Forum*, 1993.

[CLLi73] C.L.Liu and J.W.Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J.ACM*, pp. 46–61, 1973.

[Cont02] Gilberto Contreras and Patricia Nava, "Deisgn, implementation and testing of an fpga-based neuro-coprocessor," In *Posters on the Hill*, Washingtom, D.C., 2002.

[CS92] Hamilton A. et al. Churcher S., Baxter D.J., "Generic analog neural computation-the epsilon chip," In *Proceedings of the 1992 Conference of Advances in Neural Information Processing Systems*, Denver, Colorado, 1992.

[DBec92] R.K.Singh D.Becker and S.G.Tell, "An engineering environment for hardware/software cosimulation," IEEE CS Press, pp. 129–134, 1992.

[dG02] Hugo de Garis and Michael Korkin, "The cam-brain machine (cbm) an fpga based hardware tool which evolves a 1000 neuron net circuit module in seconds and updates a 75 million neuron artificial brain for real time robot control," *Neurocomputing journal*, vol. 42, , 2002.

[DGB88] P. J. Lloyd D. G. Bounds and G. Waddell, "A Multilayer Perceptron Neural Network for the Diagnosis of Low Back Pain," *IEEE International Conference on Neural Networks*, vol. 2, pp. 481–490, 1988.

[ea94] J.Buck et al., "A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation*, pp. 155–182, 1994.

[EC88] S. Ghosh E. Collins and C. Scofield, "An Application of a Multiple Neural Network Learning System to Emulation of Mortgage Underwriting Judgements," *IEEE International Conference on Neural Networks*, vol. 2, pp. 459–466, 1988.

[Eldr94] J. G. Eldredge, *Fpga density enhancement of a neural network through run-time reconfiguration* Master's thesis, Brigham Young University, 1994.

[Etem97] K. Etemad and R. Chellappa, "Discriminant analysis for recognition of human face images," pp. 1724–1733, 1997.

[Ferr94] Aaron Ferrucci, *Acme: A field-programmable gate array implementation of a self-adapting and scalable connectionist network* Master's thesis, University of California, 1994.

[GHML98] Leal Ascencio R.R. Galindo Hernandex Miriam L. and Aguilera Galicia Cuauhtemoc, "An artificial neural network on a complex programmable logic devices as a virtual sensor," Technical Report, Mexico, 1998.

[H90] Tam S. Gupta B. Castro H and Holler M., "Learning on an analog vlsi neural network chip," In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 1990.

[H94] Withagen H., In *Proceedings of the IEEE ICNN-94-Orlando Florida*, pp. 2015–2017, 1994.

[Hayk99] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1999.

[HdG97] Felix Gers Hugo de Garis and Michael Korkin, "Codi-1bit: A simplified cellular automata based neuron model," *Arificial Evolution COnference (AE97)*, 1997.

[Henn95] John L Hennessy and David A Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, INC, San Francisco, California, 1995.

[Hika03] H. Hikawa, "A new digital pulse-mode neuron with adjustable activation function," *IEEE Transactions on Neural Networks*, vol. 14, pp. 236–242, 2003.

[Hika99] H. Hikawa, "Implementation of simplified multilayer neural network with on-chip learning," In *Proc. of the IEEE International Conference on Neural Networks*, pp. 1633–1637, 1999.

[Holt91] J.L Holt and T.E Baker, "Backpropagation simulations using limited precision calculations," *International Joint Conference on Neural Networks (IJCNN-91)*, vol. 2, pp. 121–126, July 1991.

[httpa] "http://docs.gimp.org/en/index.html," .

[httpb] "http://iteso.mx/ rleal/archivos/anninfgpa.pdf," .

[httpc] "http://netpbm.sourceforge.net/doc/pgm.html," .

[httpd] "http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/avrim/ www/ml94/ face_homework.html," .

[httpe] "http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html," .

[httpf] "http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/ 9a7afa74dad200d087256ab30005f0c8/$file/opbbus.pdf," .

[httpg] "http://www.xilinx.com/bvdocs/appnotes/xapp136.pdf," .

[httph] "http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_timer.pdf," .

[httpi] "http://www.xilinx.com/ipcenter/processor_central/microblaze/ doc/hwref.pdf," .

[httpj] "http://www.xilinx.com/ipcenter/processor_central/microblaze/ doc/swref.pdf," .

[httpk] "http://www.xilinx.com/ise/embedded/est_guide.pdf," .

[httpl] "http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze," .

[JA96] Roth U. Jahnke A. and Klar H., "A simd/dataflow architecture for a neurocomputer for spike-processing neural networks," pp. 232–237, 1996.

[JB98] J.O. Haenni J.L. Beuchat and E. Sanchez, "Hardware reconfigurable neural networks," *5th Reconfigurable Architectures Workshop (RAW'98)*, March 30 1998.

[Jord95] M.I. Jordan, "Why the logistic function? A tutorial discussion on probabilities and neural networks," *MIT Computational Cognitive Science Report 9503*, vol. , pp. , 1995.

[KNic02] Medhat A. Moussa K.Nichols and S.Areibi, "Feasibility of floating-point arithmetic in fpga based artificial neural networks," *CAINE*, 2002.

[Lawr97] A.D. Lawrence, S.; C.L.; Ah Chung Tsoi; Back, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, pp. 98–113, Jan 1997.

[Ligo98] W.B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K.D. Underwood, "A re-evaluation of the practicality of floating point operations on FPGAs," In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 206–215, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[Lin97] Shang-Hung Lin; Sun-Yuan Kung; Long-Ji Lin, "Face recognition/detection by probabilistic decision-based neural network," *IEEE Transactions on Neural Networks*, pp. 114–132, Jan 1997.

[Mart94] Marcelo H. Martin, *A reconfigurable hardware accelerator for back-propagation connectionist classifiers* Master's thesis, University of California, 1994.

[McCa91] H. McCartor, "A highly parallel digital architecture for neural network emulation," *VLSI for Artificial Intelligence and Neural Networks*, Plenum Press, pp. 357–366, 1991.

[Micr] "Microblaze^{TM} risc 32-bit soft processor
xilinx inc., 2002.
http://www.xilinx.com/ipcenter/catalog/logicore/docs/
microblaze_risc_32bit_proc_final.pdf," .

[Mitc97] Tom Mitchell, *Machine Learning*, McGraw Hill, 1997.

[MY93] Hirano H. Maeda Y. and Kanata Y., "An analog neural network circuit with a learning rule via simutaneous perturbation," In *Proceedings of the IJCNN-93-Nagoya*, pp. 853–856, 1993.

[Nord95] Tomas Nordstrom, *Highly Parallel Computers for Artificial Neural Networks* PhD thesis, Division of Computer Science and Engineering, Lulea University of Technology, Sweden, March 1995.

[PT99] D. Khawparisuth P. Temdee and K. Chamnomgthai, "Face recognition by using fractal encoding and backpropagation neural network," pp. 159–161, 1999.

[PU99a] A. Perez-Uribe, *Structure-Adaptable Digital Neural Networks* PhD thesis, Logic Systems Laboratory, Computer Science Department, Swiss Federal Institute of Technology-Lausanne, October 1999.

[PU99b] Andres Perez-Uribe, *Structure-Adaptable Digital Neural Networks* PhD thesis, Logic Systems Laboratory, Computer Science Department, Swiss Federal Institute of Technology-Lausanne, 1999.

[RErn93] J.Henkel R.Ernst and T.Benner, "Hardware/Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, 1993.

[Reyn99] L. M. Reyneri, "Theoretical and implementation aspects of pulse streams: an overview," *Proceedings of the Seventh International Conference on Microelectronics for Neurla, Fuzzy and Bio-Inspired Systems*, pp. 78–89, 1999.

[RKGu93] R.K.Gupta and G.De Micheli, "Hardware/Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, pp. 29–41, 1993.

[Rosa94] Ryan Rosandich, *Artificial Vision: Three-Dimensional Object Recognition Using Neural Networks* PhD thesis, Engineering Management Department, University of Missouri-Rolla, 1994.

[RU93] Anlauf J. et al. Ramacher U., Raab W., "Multiprocessor and memory architecture of the neurocomputers synapse-1.," In *Proceedings of the 3rd International Conference on Microelectronics for Neural Networks*, pp. 227–231, 1993.

[Rume86] D.E Rumelhart, J.L McClelland, and PDP Research Group, *Parallel Distrubuted Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations, MIT Press, Cambridge, Massachusetts, 1986.

[Skrb99] M. Skrbek, "Fast neural network implementation," *Neural Network World*, Elsevier, vol. Vol. 9, No. No. 5, pp. 375–391, 1999.

[SM00] Rocio Reynoso Selene Maya and Cesar Torres, "Compact spiking neural network implementation in fpga," In *Field Programmable Logic Conference (FPL'2000)*, Villach, Austria, 2000.

[Song97] Seong-Whan Lee; Hee-Heon Song, "A new recurrent neural-network architecuture for visual pattern recognition," *IEEE Transactions on Neural Networks*, vol. 8, pp. 331–340, March 1997.

[SPra92] S.Prakash and A.C.Parker, "SOS:Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 338–351, 1992.

[Tave95] Mikael Taveniku and Arne Linde, *A reconfigurable simd computer for artificial neural networks* PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1995.

[Toh02] Meng Joo Er; Shiqian Wu; Juwei Lu; Hock Lye Toh, "Face recognition with radial basis function (RBF) neural networks," *IEEE Transactions on Neural Networks*, pp. 697–710, May 2002.

[Turk91] M. Turk and A.P. Pentland, "Face recognition using eigenfaces," vol. 3, No. 1, pp. 71–86, 1991.

[TY98] T-Y.Yen and W.Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems," *IEEE Trans. Parallel and Distributed Systems*, pp. 1125–1136, 1998.

[Wisk99] Laurenz Wiskott, Jean-Marc Fellous, Norbert Krüger, and Christoph von der Malsburg, "Face recognition by elastic bunch graph matching," In L. C. Jain, U. Halici, I. Hayashi, and S. B. Lee, editors, *Intelligent Biometric Techniques in Fingerprint and Face Recognition*, chapter 11, pp. 355–396, CRC Press, 1999.

[XL] Medhat Moussa Xiaoguang LI and Shawki Areibi, "Arithmetic formats for implementing mlp-bp on fpgas," vol. , No. , pp. .

[Yao99] Xin Yao and Tetsuya Higuchi, "Promises and challenges of evolvable hardware," *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 29, pp. 87–97, 1999.

[YT95] S.Malik Y-T.Li and A.Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, pp. 380–387, 1995.